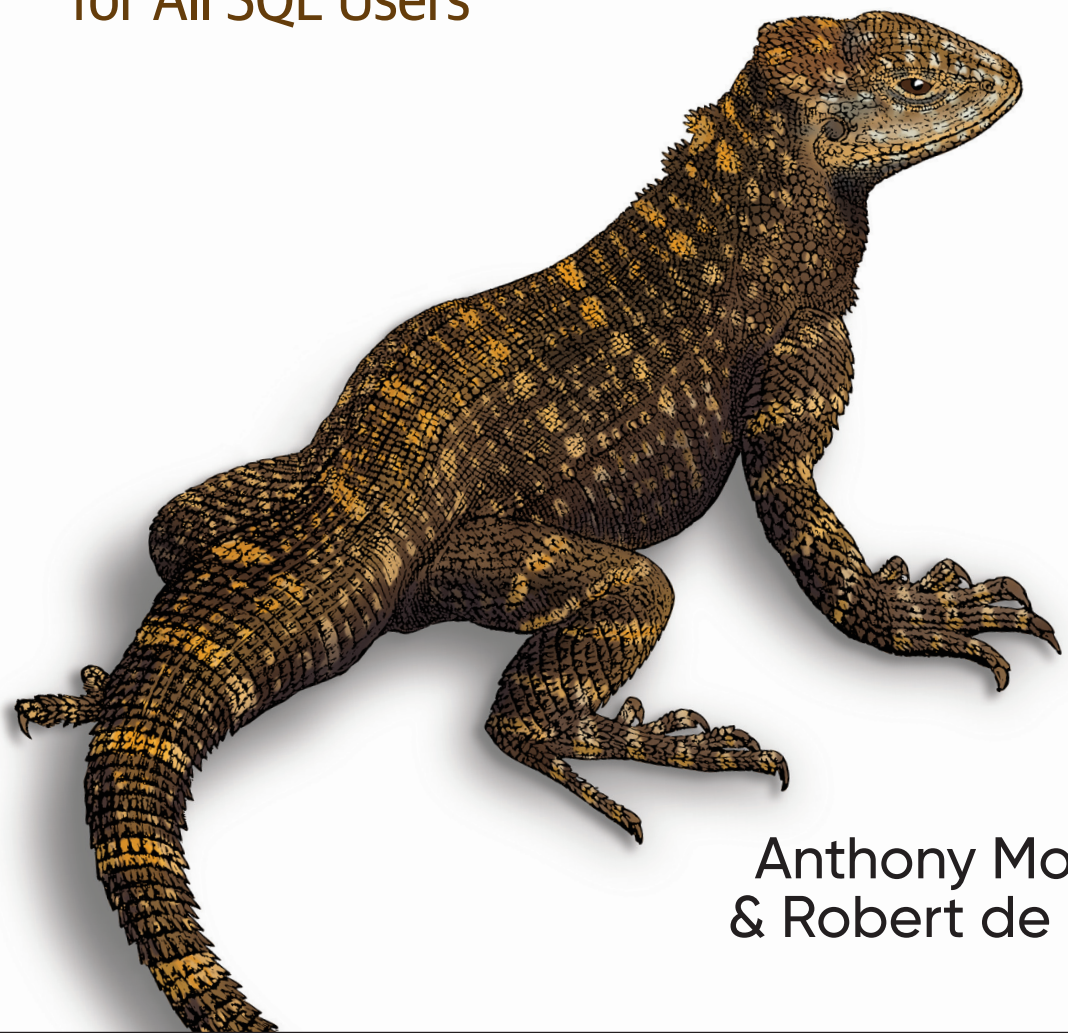
O'REILLY®

Second Edition

SQL Cookbook

Query Solutions and Techniques for All SQL Users



Anthony Molinaro
& Robert de Graaf

O'REILLY®

SQL Cookbook

You may know SQL basics, but are you taking advantage of its expressive power? This second edition applies a highly practical approach to Structured Query Language (SQL) so you can create and manipulate large stores of data. Based on real-world examples, this updated cookbook provides a framework to help you construct solutions and executable examples in several flavors of SQL, including Oracle, DB2, SQL Server, MySQL, and PostgreSQL.

SQL programmers, analysts, data scientists, database administrators—and even relatively casual SQL users—will find *SQL Cookbook* to be a valuable problem-solving guide for everyday issues. No other resource offers recipes in this unique format to help you tackle nagging day-to-day conundrums with SQL.

The second edition includes:

- Fully revised recipes that recognize the greater adoption of window functions in SQL implementations
- Additional recipes that reflect the widespread adoption of common table expressions (CTEs) for more readable, easier-to-implement solutions
- New recipes to make SQL more useful for people who aren't database experts, including data scientists
- Expanded solutions for working with numbers and strings
- Up-to-date SQL recipes throughout the book to guide you through the basics

"It's great to see a SQL cookbook updated for modern SQL topics, including window functions, common table expressions, and recursive hierarchical queries."

—**Thomas Nield**
Author of *Getting Started with SQL* (O'Reilly)

"Anthony and Robert bring a level of excitement I have yet to see in any book covering SQL. Their delivery of effective and efficient solutions is well paced, and it reinforces and complements earlier lessons learned."

—**Scott Haines**
Senior Principal Software Engineer, Twilio

Anthony Molinaro is a data scientist at Johnson & Johnson. His primary areas of research include nonparametric methods, time series analysis, and large-scale database characterization and transformation.

Robert de Graaf is senior data scientist at RightShip.

DATABASE / SQL

US $59.99 CAN $79.99
ISBN: 978-1-492-07744-2

Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

SQL Cookbook

Query Solutions and Techniques for All SQL Users

Anthony Molinaro and Robert de Graaf

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SQL Cookbook

by Anthony Molinaro and Robert de Graaf

Copyright © 2021 Robert de Graaf. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Virginia Wilson
Production Editor: Kate Galloway
Copyeditor: Kim Wimpsett
Proofreader: nSight, Inc.
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: O'Reilly Media

December 2005: First Edition
December 2020: Second Edition

Revision History for the Second Edition

2020-11-03: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492077442* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *SQL Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Yugabyte. See our statement of editorial independence.

978-1-492-07744-2

[LSI]

To my mom: You're the best! Thank you for everything.

—Anthony

To Clare, Maya, and Leda.

—Robert

Table of Contents

Preface ..... **xi**

1. Retrieving Records **1**
1.1 Retrieving All Rows and Columns from a Table 1
1.2 Retrieving a Subset of Rows from a Table 2
1.3 Finding Rows That Satisfy Multiple Conditions 2
1.4 Retrieving a Subset of Columns from a Table 3
1.5 Providing Meaningful Names for Columns 4
1.6 Referencing an Aliased Column in the WHERE Clause 5
1.7 Concatenating Column Values 6
1.8 Using Conditional Logic in a SELECT Statement 7
1.9 Limiting the Number of Rows Returned 8
1.10 Returning n Random Records from a Table 10
1.11 Finding Null Values 11
1.12 Transforming Nulls into Real Values 12
1.13 Searching for Patterns 13
1.14 Summing Up 14

2. Sorting Query Results **15**
2.1 Returning Query Results in a Specified Order 15
2.2 Sorting by Multiple Fields 16
2.3 Sorting by Substrings 17
2.4 Sorting Mixed Alphanumeric Data 18
2.5 Dealing with Nulls When Sorting 21
2.6 Sorting on a Data-Dependent Key 27
2.7 Summing Up 28

3. Working with Multiple Tables. 29
3.1 Stacking One Rowset atop Another 29
3.2 Combining Related Rows 31
3.3 Finding Rows in Common Between Two Tables 33
3.4 Retrieving Values from One Table That Do Not Exist in Another 34
3.5 Retrieving Rows from One Table That Do Not Correspond to Rows in Another 40
3.6 Adding Joins to a Query Without Interfering with Other Joins 42
3.7 Determining Whether Two Tables Have the Same Data 44
3.8 Identifying and Avoiding Cartesian Products 51
3.9 Performing Joins When Using Aggregates 52
3.10 Performing Outer Joins When Using Aggregates 57
3.11 Returning Missing Data from Multiple Tables 60
3.12 Using NULLs in Operations and Comparisons 64
3.13 Summing Up 65

4. Inserting, Updating, and Deleting. 67
4.1 Inserting a New Record 68
4.2 Inserting Default Values 68
4.3 Overriding a Default Value with NULL 70
4.4 Copying Rows from One Table into Another 70
4.5 Copying a Table Definition 71
4.6 Inserting into Multiple Tables at Once 72
4.7 Blocking Inserts to Certain Columns 74
4.8 Modifying Records in a Table 75
4.9 Updating When Corresponding Rows Exist 77
4.10 Updating with Values from Another Table 78
4.11 Merging Records 81
4.12 Deleting All Records from a Table 83
4.13 Deleting Specific Records 83
4.14 Deleting a Single Record 84
4.15 Deleting Referential Integrity Violations 85
4.16 Deleting Duplicate Records 85
4.17 Deleting Records Referenced from Another Table 87
4.18 Summing Up 89

5. Metadata Queries. 91
5.1 Listing Tables in a Schema 91
5.2 Listing a Table's Columns 93
5.3 Listing Indexed Columns for a Table 94
5.4 Listing Constraints on a Table 95
5.5 Listing Foreign Keys Without Corresponding Indexes 97

5.6 Using SQL to Generate SQL 100
5.7 Describing the Data Dictionary Views in an Oracle Database 102
5.8 Summing Up 103

6. Working with Strings. 105
6.1 Walking a String 106
6.2 Embedding Quotes Within String Literals 108
6.3 Counting the Occurrences of a Character in a String 109
6.4 Removing Unwanted Characters from a String 110
6.5 Separating Numeric and Character Data 112
6.6 Determining Whether a String Is Alphanumeric 116
6.7 Extracting Initials from a Name 120
6.8 Ordering by Parts of a String 125
6.9 Ordering by a Number in a String 126
6.10 Creating a Delimited List from Table Rows 132
6.11 Converting Delimited Data into a Multivalued IN-List 136
6.12 Alphabetizing a String 141
6.13 Identifying Strings That Can Be Treated as Numbers 147
6.14 Extracting the nth Delimited Substring 153
6.15 Parsing an IP Address 160
6.16 Comparing Strings by Sound 162
6.17 Finding Text Not Matching a Pattern 164
6.18 Summing Up 167

7. Working with Numbers. 169
7.1 Computing an Average 169
7.2 Finding the Min/Max Value in a Column 171
7.3 Summing the Values in a Column 173
7.4 Counting Rows in a Table 175
7.5 Counting Values in a Column 177
7.6 Generating a Running Total 178
7.7 Generating a Running Product 179
7.8 Smoothing a Series of Values 181
7.9 Calculating a Mode 182
7.10 Calculating a Median 185
7.11 Determining the Percentage of a Total 187
7.12 Aggregating Nullable Columns 190
7.13 Computing Averages Without High and Low Values 191
7.14 Converting Alphanumeric Strings into Numbers 193
7.15 Changing Values in a Running Total 196
7.16 Finding Outliers Using the Median Absolute Deviation 197
7.17 Finding Anomalies Using Benford's Law 201

7.18 Summing Up 203

8. Date Arithmetic. 205
8.1 Adding and Subtracting Days, Months, and Years 205
8.2 Determining the Number of Days Between Two Dates 208
8.3 Determining the Number of Business Days Between Two Dates 210
8.4 Determining the Number of Months or Years Between Two Dates 215
8.5 Determining the Number of Seconds, Minutes, or Hours Between Two Dates 218
8.6 Counting the Occurrences of Weekdays in a Year 220
8.7 Determining the Date Difference Between the Current Record and the Next Record 231
8.8 Summing Up 237

9. Date Manipulation. 239
9.1 Determining Whether a Year Is a Leap Year 240
9.2 Determining the Number of Days in a Year 246
9.3 Extracting Units of Time from a Date 249
9.4 Determining the First and Last Days of a Month 252
9.5 Determining All Dates for a Particular Weekday Throughout a Year 255
9.6 Determining the Date of the First and Last Occurrences of a Specific Weekday in a Month 261
9.7 Creating a Calendar 268
9.8 Listing Quarter Start and End Dates for the Year 281
9.9 Determining Quarter Start and End Dates for a Given Quarter 286
9.10 Filling in Missing Dates 293
9.11 Searching on Specific Units of Time 301
9.12 Comparing Records Using Specific Parts of a Date 302
9.13 Identifying Overlapping Date Ranges 305
9.14 Summing Up 311

10. Working with Ranges. 313
10.1 Locating a Range of Consecutive Values 313
10.2 Finding Differences Between Rows in the Same Group or Partition 317
10.3 Locating the Beginning and End of a Range of Consecutive Values 323
10.4 Filling in Missing Values in a Range of Values 326
10.5 Generating Consecutive Numeric Values 330
10.6 Summing Up 333

11. Advanced Searching. 335
11.1 Paginating Through a Result Set 335
11.2 Skipping n Rows from a Table 338

11.3 Incorporating OR Logic When Using Outer Joins 339
11.4 Determining Which Rows Are Reciprocals 341
11.5 Selecting the Top n Records 343
11.6 Finding Records with the Highest and Lowest Values 344
11.7 Investigating Future Rows 345
11.8 Shifting Row Values 347
11.9 Ranking Results 350
11.10 Suppressing Duplicates 351
11.11 Finding Knight Values 353
11.12 Generating Simple Forecasts 359
11.13 Summing Up 367

12. Reporting and Reshaping. 369
12.1 Pivoting a Result Set into One Row 369
12.2 Pivoting a Result Set into Multiple Rows 372
12.3 Reverse Pivoting a Result Set 377
12.4 Reverse Pivoting a Result Set into One Column 379
12.5 Suppressing Repeating Values from a Result Set 382
12.6 Pivoting a Result Set to Facilitate Inter-Row Calculations 384
12.7 Creating Buckets of Data, of a Fixed Size 386
12.8 Creating a Predefined Number of Buckets 388
12.9 Creating Horizontal Histograms 390
12.10 Creating Vertical Histograms 392
12.11 Returning Non-GROUP BY Columns 394
12.12 Calculating Simple Subtotals 397
12.13 Calculating Subtotals for All Possible Expression Combinations 400
12.14 Identifying Rows That Are Not Subtotals 410
12.15 Using Case Expressions to Flag Rows 412
12.16 Creating a Sparse Matrix 414
12.17 Grouping Rows by Units of Time 416
12.18 Performing Aggregations over Different Groups/Partitions Simultaneously 420
12.19 Performing Aggregations over a Moving Range of Values 422
12.20 Pivoting a Result Set with Subtotals 429
12.21 Summing Up 434

13. Hierarchical Queries. 435
13.1 Expressing a Parent-Child Relationship 436
13.2 Expressing a Child-Parent-Grandparent Relationship 440
13.3 Creating a Hierarchical View of a Table 444
13.4 Finding All Child Rows for a Given Parent Row 449
13.5 Determining Which Rows Are Leaf, Branch, or Root Nodes 450

13.6 Summing Up 458

14. Odds 'n' Ends. 459
14.1 Creating Cross-Tab Reports Using SQL Server's PIVOT Operator 459
14.2 Unpivoting a Cross-Tab Report Using SQL Server's UNPIVOT Operator 461
14.3 Transposing a Result Set Using Oracle's MODEL Clause 463
14.4 Extracting Elements of a String from Unfixed Locations 467
14.5 Finding the Number of Days in a Year (an Alternate Solution for Oracle) 470
14.6 Searching for Mixed Alphanumeric Strings 472
14.7 Converting Whole Numbers to Binary Using Oracle 474
14.8 Pivoting a Ranked Result Set 477
14.9 Adding a Column Header into a Double Pivoted Result Set 481
14.10 Converting a Scalar Subquery to a Composite Subquery in Oracle 493
14.11 Parsing Serialized Data into Rows 495
14.12 Calculating Percent Relative to Total 500
14.13 Testing for Existence of a Value Within a Group 502
14.14 Summing Up 505

A. Window Function Refresher. 507

B. Common Table Expressions. 535

Index. 539

Preface

SQL is the lingua franca of the data professional. At the same time, it doesn't always get the attention it deserves compared to the hot tool du jour. As result, it's common to find people who use SQL frequently but rarely or never go beyond the simplest queries, often enough because they believe that's all there is.

This book shows how much SQL can do, expanding users' tool boxes. By the end of the book you will have seen how SQL can be used for statistical analysis; to do reporting in a manner similar to Business Intelligence tools; to match text data; to perform sophisticated analysis on date data; and much more.

The first edition of *SQL Cookbook* has been a popular choice as the "second book on SQL"—the book people read after they learn the basics—since its original release. It has many strengths, such as its wide range of topics and its friendly style.

However, computing is known to move fast, even when it comes to something as mature as SQL, which has roots going back to the 1970s. While this new edition doesn't cover brand new language features, an important change is that features that were novel at the time of the first edition, and found in some implementations and not in others, are now stabilized and standardized. As a result, we have a lot more scope for developing standard solutions than was possible earlier.

There are two key examples that are important to highlight. Common table expressions (CTEs), including recursive CTEs, were available in a couple of implementations at the time the first edition was released, but are now available in all five. They were introduced to solve some practical limitations of SQL, some of which can be seen directly in these recipes. A new appendix on recursive CTEs in this edition underlines their importance and explains their relevance.

Window functions were also new enough at the time of the first edition's release that they weren't available in every implementation. They were also new enough that a special appendix was written to explain them, which remains. Now, however, window functions are in all implementations in this book. They are also in every other SQL

implementation that we're aware of, although there are so many databases out there, it's impossible to guarantee there isn't one that neglects window functions and/or CTEs.

In addition to standardizing queries where possible, we've brought new material into Chapters 6 and 7. The material in Chapter 7 unlocks new data analysis applications in recipes about the median absolute deviation and Benford's law. In Chapter 6, we have a new recipe to help match data by the sound of the text, and we have moved material on regular expressions to Chapter 6 from Chapter 14.

Who This Book Is For

This book is meant to be for any SQL user who wants to take their queries further. In terms of ability, it's meant for someone who knows at least some SQL—you might have read Alan Beaulieu's *Learning SQL*, for example—and ideally you've had to write queries on data in the wild to answer a real-life problem.

Other than those loose parameters, this is a book for all SQL users, including data engineers, data scientists, data visualization folk, BI people, etc. Some of these users may never or rarely access databases directly, but use their data visualization, BI, or statistical tool to query and fetch data. The emphasis is on practical queries that can solve real-world problems. Where a small amount of theory appears, it's there to directly support the practical elements.

What's Missing from This Book

This is a practical book, chiefly about using SQL to understand data. It doesn't cover theoretical aspects of databases, database design, or the theory behind SQL except where needed to explain specific recipes or techniques.

It also doesn't cover extensions to databases to handle data types such as XML and JSON. There are other resources available for those specialist topics.

Platform and Version

SQL is a moving target. Vendors are constantly pumping new features and functionality into their products. Thus, you should know up front which versions of the various platforms were used in the preparation of this text:

- DB2 11.5
- Oracle Database 19c
- PostgreSQL 12

- SQL Server 2017
- MySQL 8.0

Tables Used in This Book

The majority of the examples in this book involve the use of two tables, EMP and DEPT. The EMP table is a simple 14-row table with only numeric, string, and date fields. The DEPT table is a simple four-row table with only numeric and string fields. These tables appear in many old database texts, and the many-to-one relationship between departments and employees is well understood.

All but a very few solutions in this book run against these tables. Nowhere do we tweak the example data to set up a solution that you would be unlikely to have a chance of implementing in the real world, as some books do.

The contents of EMP and DEPT are shown here, respectively:

```
 select * from emp;

EMPNO ENAME  JOB        MGR HIREDATE     SAL COMM  DEPTNO
----- ------ --------- ---- ----------- ---- ---- -------
 7369 SMITH  CLERK     7902 17-DEC-2005  800           20
 7499 ALLEN  SALESMAN  7698 20-FEB-2006 1600  300      30
 7521 WARD   SALESMAN  7698 22-FEB-2006 1250  500      30
 7566 JONES  MANAGER   7839 02-APR-2006 2975           20
 7654 MARTIN SALESMAN  7698 28-SEP-2006 1250 1400      30
 7698 BLAKE  MANAGER   7839 01-MAY-2006 2850           30
 7782 CLARK  MANAGER   7839 09-JUN-2006 2450           10
 7788 SCOTT  ANALYST   7566 09-DEC-2007 3000           20
 7839 KING   PRESIDENT      17-NOV-2006 5000           10
 7844 TURNER SALESMAN  7698 08-SEP-2006 1500    0      30
 7876 ADAMS  CLERK     7788 12-JAN-2008 1100           20
 7900 JAMES  CLERK     7698 03-DEC-2006  950           30
 7902 FORD   ANALYST   7566 03-DEC-2006 3000           20
 7934 MILLER CLERK     7782 23-JAN-2007 1300           10


  select * from dept;

DEPTNO DNAME          LOC
------ -------------- ---------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

Additionally, you will find four pivot tables used in this book: T1, T10, T100, and T500. Because these tables exist only to facilitate pivots, we didn't give them clever names. The number following the "T" in each of the pivot tables signifies the number of rows in each table, starting from 1. For example, here are the values for T1 and T10:

```
select id from t1;

        ID
----------
         1

select id from t10;

        ID
----------
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10
```

The pivot tables are a useful shortcut when we need to create a series of rows to facilitate a query.

As an aside, some vendors allow partial SELECT statements. For example, you can have SELECT without a FROM clause. Sometimes in this book we will use a support table, T1, with a single row, rather than using partial queries for clarity. This is similar in usage to Oracle's DUAL table, but by using the T1 table, we do the same thing in a standardized way across all the implementations we are looking at.

Any other tables are specific to particular recipes and chapters and will be introduced in the text when appropriate.

Conventions Used in This Book

We use a number of typographical and coding conventions in this book. Take time to become familiar with them. Doing so will enhance your understanding of the text. Coding conventions in particular are important, because we can't repeat them for each recipe in the book. Instead, we list the important conventions here.

Typographical Conventions

The following typographical conventions are used in this book:

UPPERCASE
: Used to indicate SQL keywords within text.

lowercase
: Used for all queries in code examples. Other languages such as C and Java use lowercase for most keywords, and we find it far more readable than uppercase. Thus, all queries will be lowercase.

`Constant width bold`
: Indicates user input in examples showing an interaction.

Indicates a tip, suggestion, or general note.

Indicates a warning or caution.

Coding Conventions

Our preference for case in SQL statements is to always use lowercase, for both keywords and user-specified identifiers. For example:

```
select empno, ename
  from emp;
```

Your preference may be otherwise. For example, many prefer to uppercase SQL keywords. Use whatever coding style you prefer, or whatever your project requires.

Despite the use of lowercase in code examples, we consistently use uppercase for SQL keywords and identifiers in the text. We do this to make those items stand out as something other than regular prose. For example:

> The preceding query represents a SELECT against the EMP table.

While this book covers databases from five different vendors, we've decided to use one format for all the output:

```
EMPNO ENAME
----- ------
 7369 SMITH
 7499 ALLEN
  …
```

Many solutions make use of *inline views*, or subqueries in the FROM clause. The ANSI SQL standard requires that such views be given table aliases. (Oracle is the only vendor that lets you get away without specifying such aliases.) Thus, our solutions use aliases such as X and Y to identify the result sets from inline views:

```
 select job, sal
from (select job, max(sal) sal
        from emp
      group by job)x;
```

Notice the letter X following the final, closing parenthesis. That letter X becomes the name of the "table" returned by the subquery in the FROM clause. While column aliases are a valuable tool for writing self-documenting code, aliases on inline views (for most recipes in this book) are simply formalities. They are typically given trivial names such as X, Y, Z, TMP1, and TMP2. In cases where a better alias might provide more understanding, we use them.

You will notice that the SQL in the "Solution" section of the recipes is typically numbered, for example:

```
1 select ename
2     from emp
3  where deptno = 10
```

The number is not part of the syntax; it is just to reference parts of the query by number in the "Discussion" section.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/sql-ckbk-2e*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Second Edition Acknowledgments

A bunch of great people have helped with this second edition. Thanks to Jess Haberman, Virginia Wilson, Kate Galloway, and Gary O'Brien at O'Reilly. Thanks to Nicholas Adams for repeatedly saving the day in Atlas. Many thanks to the tech reviewers: Alan Beaulieu, Scott Haines, and Thomas Nield.

Finally, many thanks to my family—Clare, Maya, and Leda—for graciously bearing losing me to another book for a while.

—Robert de Graaf

First Edition Acknowledgments

This book would not exist without all the support we've received from a great many people. I would like to thank my mother, Connie, to whom this book is dedicated. Without your hard work and sacrifice, I would not be where I am today. Thank you for everything, Mom. I am thankful and appreciative of everything you've done for my brother and me. I have been blessed to have you as my mother.

To my brother, Joe: Every time I came home from Baltimore to take a break from writing, you were there to remind me how great things are when we're not working, and how I should finish writing so I can get back to the more important things in life. You're a good man, and I respect you. I am extremely proud of you, and proud to call you my brother.

To my wonderful fiancée, Georgia: Without your support I would not have made it through all 600-plus pages of this book. You were here sharing this experience with me, day after day. I know it was just as hard on you as it was on me. I spent all day working and all night writing, but you were great through it all. You were understanding and supportive, and I am forever grateful. Thank you. I love you.

To my future in-laws: To my mother-in-law and father-in-law, Kiki and George, thank you for your support throughout this whole experience. You always made me feel at home whenever I took a break and came to visit, and you made sure Georgia and I were always well fed. To my sister-in-laws, Anna and Kathy, it was always fun coming home and hanging out with you guys, giving Georgia and I a much needed break from the book and from Baltimore.

To my editor, Jonathan Gennick, without whom this book would not exist: Jonathan, you deserve a tremendous amount of credit for this book. You went above and beyond what an editor would normally do, and for that you deserve much thanks. From supplying recipes to tons of rewrites to keeping things humorous despite oncoming deadlines, I could not have done it without you. I am grateful to have had you as my editor and grateful for the opportunity you have given me. An experienced DBA and author yourself, it was a pleasure to work with someone of your technical level and expertise. I can't imagine there are too many editors out there who can, if they decided to, stop editing and work practically anywhere as a database administrator (DBA); Jonathan can. Being a DBA certainly gives you an edge as an editor as you usually know what I want to say even when I'm having trouble expressing it. O'Reilly is lucky to have you on staff, and I am lucky to have you as an editor.

I would like to thank Ales Spetic and Jonathan Gennick for *Transact-SQL Cookbook*. Isaac Newton famously said, "If I have seen a little further it is by standing on the shoulders of giants." In the acknowledgments section of the *Transact-SQL Cookbook*, Ales Spetic wrote something that is a testament to this famous quote, and I feel should be in every SQL book. I include his words here:

> I hope that this book will complement the exiting opuses of outstanding authors like Joe Celko, David Rozenshtein, Anatoly Abramovich, Eugine Berger, Iztik Ben-Gan, Richard Snodgrass, and others. I spent many nights studying their work, and I learned almost everything I know from their books. As I am writing these lines, I'm aware that for every night I spent discovering their secrets, they must have spent 10 nights putting their knowledge into a consistent and readable form. It is an honor to be able to give something back to the SQL community.

I would like to thank Sanjay Mishra for his excellent *Mastering Oracle SQL* book, and also for putting me in touch with Jonathan. If not for Sanjay, I may have never met Jonathan and never would have written this book. Amazing how a simple email can change your life. I would like to thank David Rozenshtein, especially, for his *Essence of SQL* book, which provided me with a solid understanding of how to think and problem solve in sets/SQL. I would like to thank David Rozenshtein, Anatoly Abramovich, and Eugene Birger for their book *Optimizing Transact-SQL*, from which I learned many of the advanced SQL techniques I use today.

I would like to thank the whole team at Wireless Generation, a great company with great people. A big thank-you to all of the people who took the time to review, critique, or offer advice to help me complete this book: Jesse Davis, Joel Patterson, Philip Zee, Kevin Marshall, Doug Daniels, Otis Gospodnetic, Ken Gunn, John Stewart, Jim Abramson, Adam Mayer, Susan Lau, Alexis Le-Quoc, and Paul Feuer. I would like to thank Maggie Ho for her careful review of my work and extremely useful feedback regarding the window function refresher. I would like to thank Chuck Van Buren and Gillian Gutenberg for their great advice about running. Early morning workouts helped me clear my mind and unwind. I don't think I would have been able to finish this book without getting out a bit. I would like to thank Steve Kang and Chad Levinson for putting up with all my incessant talk about different SQL techniques on the nights when all they wanted was to head to Union Square to get a beer and a burger at Heartland Brewery after a long day of work. I would like to thank Aaron Boyd for all his support, kind words, and, most importantly, good advice. Aaron is honest, hard-working, and a very straightforward guy; people like him make a company better. I would like to thank Olivier Pomel for his support and help in writing this book, in particular for the DB2 solution for creating delimited lists from rows. Olivier contributed that solution without even having a DB2 system to test it! I explained to him how the WITH clause worked, and minutes later he came up with the solution you see in this book.

Jonah Harris and David Rozenshtein also provided helpful technical review feedback on the manuscript. And Arun Marathe, Nuno Pinto do Souto, and Andrew Odewahn weighed in on the outline and choice of recipes while this book was in its formative stages. Thanks, very much, to all of you.

I want to thank John Haydu and the MODEL clause development team at Oracle Corporation for taking the time to review the MODEL clause article I wrote for O'Reilly, and for ultimately giving me a better understanding of how that clause works. I would like to thank Tom Kyte of Oracle Corporation for allowing me to adapt his TO_BASE function into a SQL-only solution. Bruno Denuit of Microsoft answered questions I had regarding the functionality of the window functions introduced in SQL Server 2005. Simon Riggs of PostgreSQL kept me up-to-date about new SQL features in PostgreSQL (very big thanks: Simon, by knowing what was coming out and when, I was able to incorporate some new SQL features such as the ever-so-

cool GENERATE_SERIES function, which I think made for more elegant solutions compared to pivot tables).

Last but certainly not least, I'd like to thank Kay Young. When you are talented and passionate about what you do, it is great to be able to work with people who are likewise as talented and passionate. Many of the recipes you see in this text have come from working with Kay and coming up with SQL solutions for everyday problems at Wireless Generation. I want to thank you and let you know I absolutely appreciate all the help you have given me throughout all of this; from advice to grammar corrections to code, you played an integral role in the writing of this book. It's been great working with you, and Wireless Generation is a better company because you are there.

—Anthony Molinaro

CHAPTER 1

Retrieving Records

This chapter focuses on basic SELECT statements. It is important to have a solid understanding of the basics as many of the topics covered here are not only present in more difficult recipes but are also found in everyday SQL.

1.1 Retrieving All Rows and Columns from a Table

Problem

You have a table and want to see all of the data in it.

Solution

Use the special * character and issue a SELECT against the table:

```
1 select *
2   from emp
```

Discussion

The character * has special meaning in SQL. Using it will return every column for the table specified. Since there is no WHERE clause specified, every row will be returned as well. The alternative would be to list each column individually:

```
select empno,ename,job,sal,mgr,hiredate,comm,deptno
  from emp
```

In ad hoc queries that you execute interactively, it's easier to use SELECT *. However, when writing program code, it's better to specify each column individually. The performance will be the same, but by being explicit you will always know what columns you are returning from the query. Likewise, such queries are easier to understand by

people other than yourself (who may or may not know all the columns in the tables in the query). Problems with SELECT * can also arise if your query is within code, and the program gets a different set of columns from the query than was expected. At least, if you specify all columns and one or more is missing, any error thrown is more likely to be traceable to the specific missing column(s).

1.2 Retrieving a Subset of Rows from a Table

Problem

You have a table and want to see only rows that satisfy a specific condition.

Solution

Use the WHERE clause to specify which rows to keep. For example, to view all employees assigned to department number 10:

```
1 select *
2   from emp
3  where deptno = 10
```

Discussion

The WHERE clause allows you to retrieve only rows you are interested in. If the expression in the WHERE clause is true for any row, then that row is returned.

Most vendors support common operators such as =, <, >, <=, >=, !, and <>. Additionally, you may want rows that satisfy multiple conditions; this can be done by specifying AND, OR, and parentheses, as shown in the next recipe.

1.3 Finding Rows That Satisfy Multiple Conditions

Problem

You want to return rows that satisfy multiple conditions.

Solution

Use the WHERE clause along with the OR and AND clauses. For example, if you would like to find all the employees in department 10, along with any employees who earn a commission, along with any employees in department 20 who earn at most $2,000:

```
1 select *
2   from emp
3  where deptno = 10
4     or comm is not null
5     or sal <= 2000 and deptno=20
```

Discussion

You can use a combination of AND, OR, and parentheses to return rows that satisfy multiple conditions. In the solution example, the WHERE clause finds rows such that:

- The DEPTNO is 10
- The COMM is not NULL
- The salary is $2,000 or less for any employee in DEPTNO 20.

The presence of parentheses causes conditions within them to be evaluated together.

For example, consider how the result set changes if the query was written with the parentheses as shown here:

```
select *
 from emp
where (     deptno = 10
        or comm is not null
        or sal <= 2000
      )
  and deptno=20

EMPNO ENAME  JOB     MGR  HIREDATE      SAL       COMM  DEPTNO
----- ------ ----- -----  ----------- ----- ----------  ------
 7369 SMITH  CLERK  7902  17-DEC-1980   800                 20
 7876 ADAMS  CLERK  7788  12-JAN-1983  1100                 20
```

1.4 Retrieving a Subset of Columns from a Table

Problem

You have a table and want to see values for specific columns rather than for all the columns.

Solution

Specify the columns you are interested in. For example, to see only name, department number, and salary for employees:

```
1 select ename,deptno,sal
2   from emp
```

Discussion

By specifying the columns in the SELECT clause, you ensure that no extraneous data is returned. This can be especially important when retrieving data across a network, as it avoids the waste of time inherent in retrieving data that you do not need.

1.5 Providing Meaningful Names for Columns

Problem

You would like to change the names of the columns that are returned by your query so they are more readable and understandable. Consider this query that returns the salaries and commissions for each employee:

```
1 select sal,comm
2   from emp
```

What's SAL? Is it short for *sale*? Is it someone's name? What's COMM? Is it communication? You want the results to have more meaningful labels.

Solution

To change the names of your query results, use the AS keyword in the form `original_name AS new_name`. Some databases do not require AS, but all accept it:

```
 1 select sal as salary, comm as commission
  2   from emp

SALARY   COMMISSION
-------  ----------
    800
   1600        300
   1250        500
   2975
   1250       1400
   2850
   2450
   3000
   5000
   1500          0
   1100
    950
   3000
   1300
```

Discussion

Using the AS keyword to give new names to columns returned by your query is known as *aliasing* those columns. The new names that you give are known as *aliases*.

Creating good aliases can go a long way toward making a query and its results understandable to others.

1.6 Referencing an Aliased Column in the WHERE Clause

Problem

You have used aliases to provide more meaningful column names for your result set and would like to exclude some of the rows using the WHERE clause. However, your attempt to reference alias names in the WHERE clause fails:

```
select sal as salary, comm as commission
  from emp
 where salary < 5000
```

Solution

By wrapping your query as an inline view, you can reference the aliased columns:

```
1 select *
2   from (
3 select sal as salary, comm as commission
4   from emp
5        ) x
6  where salary < 5000
```

Discussion

In this simple example, you can avoid the inline view and reference COMM or SAL directly in the WHERE clause to achieve the same result. This solution introduces you to what you would need to do when attempting to reference any of the following in a WHERE clause:

- Aggregate functions
- Scalar subqueries
- Windowing functions
- Aliases

Placing your query, the one giving aliases, in an inline view gives you the ability to reference the aliased columns in your outer query. Why do you need to do this? The WHERE clause is evaluated before the SELECT; thus, SALARY and COMMISSION do not yet exist when the "Problem" query's WHERE clause is evaluated. Those aliases are not applied until after the WHERE clause processing is complete. However, the FROM clause is evaluated before the WHERE. By placing the original query in a FROM clause, the results from that query are generated before the outermost

WHERE clause, and your outermost WHERE clause "sees" the alias names. This technique is particularly useful when the columns in a table are not named particularly well.

The inline view in this solution is aliased X. Not all databases require an inline view to be explicitly aliased, but some do. All of them accept it.

1.7 Concatenating Column Values

Problem

You want to return values in multiple columns as one column. For example, you would like to produce this result set from a query against the EMP table:

```
CLARK WORKS AS A MANAGER
KING WORKS AS A PRESIDENT
MILLER WORKS AS A CLERK
```

However, the data that you need to generate this result set comes from two different columns, the ENAME and JOB columns in the EMP table:

```
 select ename, job
    from emp
    where deptno = 10

ENAME      JOB
---------- ---------
CLARK      MANAGER
KING       PRESIDENT
MILLER     CLERK
```

Solution

Find and use the built-in function provided by your DBMS to concatenate values from multiple columns.

DB2, Oracle, PostgreSQL

These databases use the double vertical bar as the concatenation operator:

```
1 select ename||' WORKS AS A '||job as msg
2   from emp
3  where deptno=10
```

MySQL

This database supports a function called CONCAT:

```
1 select concat(ename, ' WORKS AS A ',job) as msg
2   from emp
3  where deptno=10
```

SQL Server

Use the + operator for concatenation:

```
1 select ename + ' WORKS AS A ' + job as msg
2   from emp
3  where deptno=10
```

Discussion

Use the CONCAT function to concatenate values from multiple columns. The || is a shortcut for the CONCAT function in DB2, Oracle, and PostgreSQL, while + is the shortcut for SQL Server.

1.8 Using Conditional Logic in a SELECT Statement

Problem

You want to perform IF-ELSE operations on values in your SELECT statement. For example, you would like to produce a result set such that if an employee is paid $2,000 or less, a message of "UNDERPAID" is returned; if an employee is paid $4,000 or more, a message of "OVERPAID" is returned; and if they make somewhere in between, then "OK" is returned. The result set should look like this:

```
 ENAME            SAL  STATUS
---------- ----------  ---------
SMITH            800  UNDERPAID
ALLEN           1600  UNDERPAID
WARD            1250  UNDERPAID
JONES           2975  OK
MARTIN          1250  UNDERPAID
BLAKE           2850  OK
CLARK           2450  OK
SCOTT           3000  OK
KING            5000  OVERPAID
TURNER          1500  UNDERPAID
ADAMS           1100  UNDERPAID
JAMES            950  UNDERPAID
FORD            3000  OK
MILLER          1300  UNDERPAID
```

Solution

Use the CASE expression to perform conditional logic directly in your SELECT statement:

```
1 select ename,sal,
2        case when sal <= 2000 then 'UNDERPAID'
3             when sal >= 4000 then 'OVERPAID'
4             else 'OK'
5        end as status
6   from emp
```

Discussion

The CASE expression allows you to perform condition logic on values returned by a query. You can provide an alias for a CASE expression to return a more readable result set. In the solution, you'll see the alias STATUS given to the result of the CASE expression. The ELSE clause is optional. Omit the ELSE, and the CASE expression will return NULL for any row that does not satisfy the test condition.

1.9 Limiting the Number of Rows Returned

Problem

You want to limit the number of rows returned in your query. You are not concerned with order; any *n* rows will do.

Solution

Use the built-in function provided by your database to control the number of rows returned.

DB2

In DB2 use the FETCH FIRST clause:

```
1 select *
2   from emp fetch first 5 rows only
```

MySQL and PostgreSQL

Do the same thing in MySQL and PostgreSQL using LIMIT:

```
1 select *
2   from emp limit 5
```

Oracle

In Oracle, place a restriction on the number of rows returned by restricting ROWNUM in the WHERE clause:

```
1 select *
2   from emp
3  where rownum <= 5
```

SQL Server

Use the TOP keyword to restrict the number of rows returned:

```
1 select top 5 *
2   from emp
```

Discussion

Many vendors provide clauses such as FETCH FIRST and LIMIT that let you specify the number of rows to be returned from a query. Oracle is different, in that you must make use of a function called ROWNUM that returns a number for each row returned (an increasing value starting from one).

Here is what happens when you use ROWNUM <= 5 to return the first five rows:

1. Oracle executes your query.
2. Oracle fetches the first row and calls it row number one.
3. Have we gotten past row number five yet? If no, then Oracle returns the row, because it meets the criteria of being numbered less than or equal to five. If yes, then Oracle does not return the row.
4. Oracle fetches the next row and advances the row number (to two, then to three, then to four, and so forth).
5. Go to step 3.

As this process shows, values from Oracle's ROWNUM are assigned *after* each row is fetched. This is an important and key point. Many Oracle developers attempt to return only, say, the fifth row returned by a query by specifying ROWNUM = 5.

Using an equality condition in conjunction with ROWNUM is a bad idea. Here is what happens when you try to return, say, the fifth row using ROWNUM = 5:

1. Oracle executes your query.
2. Oracle fetches the first row and calls it row number one.
3. Have we gotten to row number five yet? If no, then Oracle discards the row, because it doesn't meet the criteria. If yes, then Oracle returns the row. But the answer will never be yes!

4. Oracle fetches the next row and calls it row number one. This is because the first row to be returned from the query must be numbered as one.
5. Go to step 3.

Study this process closely, and you can see why the use of ROWNUM = 5 to return the fifth row fails. You can't have a fifth row if you don't first return rows one through four!

You may notice that ROWNUM = 1 does, in fact, work to return the first row, which may seem to contradict the explanation thus far. The reason ROWNUM = 1 works to return the first row is that, to determine whether there are any rows in the table, Oracle has to attempt to fetch at least once. Read the preceding process carefully, substituting one for five, and you'll understand why it's OK to specify ROWNUM = 1 as a condition (for returning one row).

1.10 Returning n Random Records from a Table

Problem

You want to return a specific number of random records from a table. You want to modify the following statement such that successive executions will produce a different set of five rows:

```
select ename, job
  from emp
```

Solution

Take any built-in function supported by your DBMS for returning random values. Use that function in an ORDER BY clause to sort rows randomly. Then, use the previous recipe's technique to limit the number of randomly sorted rows to return.

DB2

Use the built-in function RAND in conjunction with ORDER BY and FETCH:

```
1 select ename,job
2   from emp
3  order by rand() fetch first 5 rows only
```

MySQL

Use the built-in RAND function in conjunction with LIMIT and ORDER BY:

```
1 select ename,job
2   from emp
3  order by rand() limit 5
```

PostgreSQL

Use the built-in RANDOM function in conjunction with LIMIT and ORDER BY:

```
 1 select ename,job
2   from emp
3  order by random() limit 5
```

Oracle

Use the built-in function VALUE, found in the built-in package DBMS_RANDOM, in conjunction with ORDER BY and the built-in function ROWNUM:

```
1 select *
2   from (
3  select ename, job
4    from emp
6   order by dbms_random.value()
7        )
8   where rownum <= 5
```

SQL Server

Use the built-in function NEWID in conjunction with TOP and ORDER BY to return a random result set:

```
1 select top 5 ename,job
2   from emp
3  order by newid()
```

Discussion

The ORDER BY clause can accept a function's return value and use it to change the order of the result set. These solutions all restrict the number of rows to return *after* the function in the ORDER BY clause is executed. Non-Oracle users may find it helpful to look at the Oracle solution as it shows (conceptually) what is happening under the covers of the other solutions.

It is important that you don't confuse using a function in the ORDER BY clause with using a numeric constant. When specifying a numeric constant in the ORDER BY clause, you are requesting that the sort be done according the column in that ordinal position in the SELECT list. When you specify a function in the ORDER BY clause, the sort is performed on the result from the function as it is evaluated for each row.

1.11 Finding Null Values

Problem

You want to find all rows that are null for a particular column.

Solution

To determine whether a value is null, you must use IS NULL:

```
1 select *
2   from emp
3  where comm is null
```

Discussion

NULL is never equal/not equal to anything, not even itself; therefore, you cannot use = or != for testing whether a column is NULL. To determine whether a row has NULL values, you must use IS NULL. You can also use IS NOT NULL to find rows without a null in a given column.

1.12 Transforming Nulls into Real Values

Problem

You have rows that contain nulls and would like to return non-null values in place of those nulls.

Solution

Use the function COALESCE to substitute real values for nulls:

```
1 select coalesce(comm,0)
2   from emp
```

Discussion

The COALESCE function takes one or more values as arguments. The function returns the first non-null value in the list. In the solution, the value of COMM is returned whenever COMM is not null. Otherwise, a zero is returned.

When working with nulls, it's best to take advantage of the built-in functionality provided by your DBMS; in many cases you'll find several functions work equally as well for this task. COALESCE happens to work for all DBMSs. Additionally, CASE can be used for all DBMSs as well:

```
select case
       when comm is not null then comm
       else 0
       end
  from emp
```

While you can use CASE to translate nulls into values, you can see that it's much easier and more succinct to use COALESCE.

1.13 Searching for Patterns

Problem

You want to return rows that match a particular substring or pattern. Consider the following query and result set:

```
select ename, job
  from emp
 where deptno in (10,20)

ENAME       JOB
----------  ---------
SMITH       CLERK
JONES       MANAGER
CLARK       MANAGER
SCOTT       ANALYST
KING        PRESIDENT
ADAMS       CLERK
FORD        ANALYST
MILLER      CLERK
```

Of the employees in departments 10 and 20, you want to return only those that have either an "I" somewhere in their name or a job title ending with "ER":

```
ENAME       JOB
----------  ---------
SMITH       CLERK
JONES       MANAGER
CLARK       MANAGER
KING        PRESIDENT
MILLER      CLERK
```

Solution

Use the LIKE operator in conjunction with the SQL wildcard operator (%):

```
1 select ename, job
2   from emp
3  where deptno in (10,20)
4    and (ename like '%I%' or job like '%ER')
```

Discussion

When used in a LIKE pattern-match operation, the percent (%) operator matches any sequence of characters. Most SQL implementations also provide the underscore ("_") operator to match a single character. By enclosing the search pattern "I" with % operators, any string that contains an "I" (at any position) will be returned. If you do not enclose the search pattern with %, then where you place the operator will affect the results of the query. For example, to find job titles that end in "ER," prefix the %

operator to "ER"; if the requirement is to search for all job titles beginning with "ER," then append the % operator to "ER."

1.14 Summing Up

These recipes may be simple, but they are also fundamental. Information retrieval is the core of database querying, and that means these recipes are at the heart of virtually everything that is discussed throughout the rest of the book.

CHAPTER 2

Sorting Query Results

This chapter focuses on customizing how your query results look. By understanding how to control how your result set is organized, you can provide more readable and meaningful data.

2.1 Returning Query Results in a Specified Order

Problem

You want to display the names, jobs, and salaries of employees in department 10 in order based on their salary (from lowest to highest). You want to return the following result set:

```
ENAME       JOB               SAL
----------  ---------  ----------
MILLER      CLERK            1300
CLARK       MANAGER          2450
KING        PRESIDENT        5000
```

Solution

Use the ORDER BY clause:

```
1 select ename,job,sal
2   from emp
3  where deptno = 10
4  order by sal asc
```

Discussion

The ORDER BY clause allows you to order the rows of your result set. The solution sorts the rows based on SAL in ascending order. By default, ORDER BY will sort in

ascending order, and the ASC clause is therefore optional. Alternatively, specify DESC to sort in descending order:

```
select ename,job,sal
  from emp
 where deptno = 10
 order by sal desc

ENAME       JOB               SAL
----------  ---------  ----------
KING        PRESIDENT        5000
CLARK       MANAGER          2450
MILLER      CLERK            1300
```

You need not specify the name of the column on which to sort. You can instead specify a number representing the column. The number starts at 1 and matches the items in the SELECT list from left to right. For example:

```
select ename,job,sal
  from emp
 where deptno = 10
 order by 3 desc

ENAME       JOB               SAL
----------  ---------  ----------
KING        PRESIDENT        5000
CLARK       MANAGER          2450
MILLER      CLERK            1300
```

The number 3 in this example's ORDER BY clause corresponds to the third column in the SELECT list, which is SAL.

2.2 Sorting by Multiple Fields

Problem

You want to sort the rows from EMP first by DEPTNO ascending, then by salary descending. You want to return the following result set:

```
     EMPNO      DEPTNO         SAL  ENAME       JOB
----------  ----------  ----------  ----------  ---------
      7839          10        5000  KING        PRESIDENT
      7782          10        2450  CLARK       MANAGER
      7934          10        1300  MILLER      CLERK
      7788          20        3000  SCOTT       ANALYST
      7902          20        3000  FORD        ANALYST
      7566          20        2975  JONES       MANAGER
      7876          20        1100  ADAMS       CLERK
      7369          20         800  SMITH       CLERK
      7698          30        2850  BLAKE       MANAGER
      7499          30        1600  ALLEN       SALESMAN
```

```
7844          30        1500  TURNER      SALESMAN
7521          30        1250  WARD        SALESMAN
7654          30        1250  MARTIN      SALESMAN
7900          30         950  JAMES       CLERK
```

Solution

List the different sort columns in the ORDER BY clause, separated by commas:

```
1 select empno,deptno,sal,ename,job
2   from emp
3  order by deptno, sal desc
```

Discussion

The order of precedence in ORDER BY is from left to right. If you are ordering using the numeric position of a column in the SELECT list, then that number must not be greater than the number of items in the SELECT list. You are generally permitted to order by a column not in the SELECT list, but to do so you must explicitly name the column. However, if you are using GROUP BY or DISTINCT in your query, you cannot order by columns that are not in the SELECT list.

2.3 Sorting by Substrings

Problem

You want to sort the results of a query by specific parts of a string. For example, you want to return employee names and jobs from table EMP and sort by the last two characters in the JOB field. The result set should look like the following:

```
ENAME       JOB
----------  ---------
KING        PRESIDENT
SMITH       CLERK
ADAMS       CLERK
JAMES       CLERK
MILLER      CLERK
JONES       MANAGER
CLARK       MANAGER
BLAKE       MANAGER
ALLEN       SALESMAN
MARTIN      SALESMAN
WARD        SALESMAN
TURNER      SALESMAN
SCOTT       ANALYST
FORD        ANALYST
```

Solution

DB2, MySQL, Oracle, and PostgreSQL

Use the SUBSTR function in the ORDER BY clause:

```
select ename,job
  from emp
 order by substr(job,length(job)-1)
```

SQL Server

Use the SUBSTRING function in the ORDER BY clause:

```
select ename,job
  from emp
 order by substring(job,len(job)-1,2)
```

Discussion

Using your DBMS's substring function, you can easily sort by any part of a string. To sort by the last two characters of a string, find the end of the string (which is the length of the string) and subtract two. The start position will be the second to last character in the string. You then take all characters after that start position. SQL Server's SUBSTRING is different from the SUBSTR function as it requires a third parameter that specifies how many characters to take. In this example, any number greater than or equal to two will work.

2.4 Sorting Mixed Alphanumeric Data

Problem

You have mixed alphanumeric data and want to sort by either the numeric or character portion of the data. Consider this view, created from the EMP table:

```
create view V
as
select ename||' '||deptno as data
  from emp

select * from V

DATA
-------------
SMITH 20
ALLEN 30
WARD 30
JONES 20
MARTIN 30
```

```
BLAKE 30
CLARK 10
SCOTT 20
KING 10
TURNER 30
ADAMS 20
JAMES 30
FORD 20
MILLER 10
```

You want to sort the results by DEPTNO or ENAME. Sorting by DEPTNO produces the following result set:

```
DATA
----------
CLARK 10
KING 10
MILLER 10
SMITH 20
ADAMS 20
FORD 20
SCOTT 20
JONES 20
ALLEN 30
BLAKE 30
MARTIN 30
JAMES 30
TURNER 30
WARD 30
```

Sorting by ENAME produces the following result set:

```
DATA
---------
ADAMS 20
ALLEN 30
BLAKE 30
CLARK 10
FORD 20
JAMES 30
JONES 20
KING 10
MARTIN 30
MILLER 10
SCOTT 20
SMITH 20
TURNER 30
WARD 30
```

Solution

Oracle, SQL Server, and PostgreSQL

Use the functions REPLACE and TRANSLATE to modify the string for sorting:

```
/* ORDER BY DEPTNO */

1 select data
2   from V
3  order by replace(data,
4           replace(
5         translate(data,'0123456789','##########'),'#',''),'')

/* ORDER BY ENAME */

1 select data
2   from V
3  order by replace(
4           translate(data,'0123456789','##########'),'#','')
```

DB2

Implicit type conversion is more strict in DB2 than in Oracle or PostgreSQL, so you will need to cast DEPTNO to a CHAR for view V to be valid. Rather than re-create view V, this solution will simply use an inline view. The solution uses REPLACE and TRANSLATE in the same way as the Oracle and PostrgreSQL solution, but the order of arguments for TRANSLATE is slightly different for DB2:

```
/* ORDER BY DEPTNO */

1  select *
2    from (
3  select ename||' '||cast(deptno as char(2)) as data
4    from emp
5        ) v
6   order by replace(data,
7            replace(
8           translate(data,'##########','0123456789'),'#',''),'')

/* ORDER BY ENAME */

1  select *
2    from (
3  select ename||' '||cast(deptno as char(2)) as data
4    from emp
5        ) v
6   order by replace(
7            translate(data,'##########','0123456789'),'#','')
```

MySQL

The TRANSLATE function is not currently supported by these platforms; thus, a solution for this problem will not be provided.

Discussion

The TRANSLATE and REPLACE functions remove either the numbers or characters from each row, allowing you to easily sort by one or the other. The values passed to ORDER BY are shown in the following query results (using the Oracle solution as the example, as the same technique applies to all three vendors; only the order of parameters passed to TRANSLATE is what sets DB2 apart):

```
select data,
       replace(data,
       replace(
     translate(data,'0123456789','##########'),'#',''),'') nums,
       replace(
     translate(data,'0123456789','##########'),'#','') chars
  from V

DATA         NUMS   CHARS
------------ ------ ----------
SMITH 20     20     SMITH
ALLEN 30     30     ALLEN
WARD 30      30     WARD
JONES 20     20     JONES
MARTIN 30    30     MARTIN
BLAKE 30     30     BLAKE
CLARK 10     10     CLARK
SCOTT 20     20     SCOTT
KING 10      10     KING
TURNER 30    30     TURNER
ADAMS 20     20     ADAMS
JAMES 30     30     JAMES
FORD 20      20     FORD
MILLER 10    10     MILLER
```

2.5 Dealing with Nulls When Sorting

Problem

You want to sort results from EMP by COMM, but the field is nullable. You need a way to specify whether nulls sort last:

```
ENAME              SAL        COMM
----------  ----------  ----------
TURNER            1500           0
ALLEN             1600         300
WARD              1250         500
```

```
MARTIN            1250        1400
SMITH              800
JONES             2975
JAMES              950
MILLER            1300
FORD              3000
ADAMS             1100
BLAKE             2850
CLARK             2450
SCOTT             3000
KING              5000
```

or whether they sort first:

```
ENAME              SAL        COMM
----------  ----------  ----------
SMITH              800
JONES             2975
CLARK             2450
BLAKE             2850
SCOTT             3000
KING              5000
JAMES              950
MILLER            1300
FORD              3000
ADAMS             1100
MARTIN            1250        1400
WARD              1250         500
ALLEN             1600         300
TURNER            1500           0
```

Solution

Depending on how you want the data to look and how your particular RDBMS sorts NULL values, you can sort the nullable column in ascending or descending order:

```
1 select ename,sal,comm
2   from emp
3  order by 3

1 select ename,sal,comm
2   from emp
3  order by 3 desc
```

This solution puts you in a position such that if the nullable column contains non-NULL values, they will be sorted in ascending or descending order as well, according to what you ask for; this may or may not be what you have in mind. If instead you would like to sort NULL values differently than non-NULL values, for example, you want to sort non-NULL values in ascending or descending order and all NULL values last, you can use a CASE expression to conditionally sort the column.

DB2, MySQL, PostgreSQL, and SQL Server

Use a CASE expression to "flag" when a value is NULL. The idea is to have a flag with two values: one to represent NULLs, the other to represent non-NULLs. Once you have that, simply add this flag column to the ORDER BY clause. You'll easily be able to control whether NULL values are sorted first or last without interfering with non-NULL values:

```
/* NON-NULL COMM SORTED ASCENDING, ALL NULLS LAST */

1  select ename,sal,comm
2    from (
3  select ename,sal,comm,
4         case when comm is null then 0 else 1 end as is_null
5    from emp
6         ) x
7    order by is_null desc,comm

ENAME     SAL        COMM
------  -----  ----------
TURNER   1500           0
ALLEN    1600         300
WARD     1250         500
MARTIN   1250        1400
SMITH     800
JONES    2975
JAMES     950
MILLER   1300
FORD     3000
ADAMS    1100
BLAKE    2850
CLARK    2450
SCOTT    3000
KING     5000

/* NON-NULL COMM SORTED DESCENDING, ALL NULLS LAST */

1  select ename,sal,comm
2    from (
3  select ename,sal,comm,
4         case when comm is null then 0 else 1 end as is_null
5    from emp
6         ) x
7   order by is_null desc,comm desc

ENAME     SAL        COMM
------  -----  ----------
MARTIN   1250        1400
WARD     1250         500
ALLEN    1600         300
TURNER   1500           0
SMITH     800
```

```
JONES    2975
JAMES     950
MILLER   1300
FORD     3000
ADAMS    1100
BLAKE    2850
CLARK    2450
SCOTT    3000
KING     5000

/* NON-NULL COMM SORTED ASCENDING, ALL NULLS FIRST */

1 select ename,sal,comm
2   from (
3 select ename,sal,comm,
4        case when comm is null then 0 else 1 end as is_null
5   from emp
6        ) x
7  order by is_null,comm

ENAME    SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
TURNER  1500          0
ALLEN   1600        300
WARD    1250        500
MARTIN  1250       1400

/* NON-NULL COMM SORTED DESCENDING, ALL NULLS FIRST */

1  select ename,sal,comm
2    from (
3  select ename,sal,comm,
4         case when comm is null then 0 else 1 end as is_null
5    from emp
6         ) x
7   order by is_null,comm desc

ENAME    SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
```

```
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
MARTIN  1250       1400
WARD    1250        500
ALLEN   1600        300
TURNER  1500          0
```

Oracle

Oracle users can use the solution for the other platforms. They can also use the following Oracle-only solution, taking advantage of the NULLS FIRST and NULLS LAST extension to the ORDER BY clause to ensure NULLs are sorted first or last regardless of how non-NULL values are sorted:

```
/* NON-NULL COMM SORTED ASCENDING, ALL NULLS LAST */

1 select ename,sal,comm
2   from emp
3  order by comm nulls last

ENAME    SAL       COMM
------  ----- ---------
TURNER   1500         0
ALLEN    1600       300
WARD     1250       500
MARTIN   1250      1400
SMITH     800
JONES    2975
JAMES     950
MILLER   1300
FORD     3000
ADAMS    1100
BLAKE    2850
CLARK    2450
SCOTT    3000
KING     5000

/* NON-NULL COMM SORTED ASCENDING, ALL NULLS FIRST */

1 select ename,sal,comm
2   from emp
3  order by comm nulls first

ENAME    SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
```

```
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
TURNER  1500          0
ALLEN   1600        300
WARD    1250        500
MARTIN  1250       1400

/* NON-NULL COMM SORTED DESCENDING, ALL NULLS FIRST */

1 select ename,sal,comm
2   from emp
3  order by comm desc nulls first

ENAME    SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
MARTIN  1250       1400
WARD    1250        500
ALLEN   1600        300
TURNER  1500          0
```

Discussion

Unless your RDBMS provides you with a way to easily sort NULL values first or last without modifying non-NULL values in the same column (as Oracle does), you'll need an auxiliary column.

As of the time of this writing, DB2 users can use NULLS FIRST and NULLS LAST in the ORDER BY subclause of the OVER clause in window functions but not in the ORDER BY clause for the entire result set.

The purpose of this extra column (in the query only, not in the table) is to allow you to identify NULL values and sort them altogether, first or last. The following query returns the result set for inline view X for the non-Oracle solution:

```
select ename,sal,comm,
       case when comm is null then 0 else 1 end as is_null
  from emp

ENAME    SAL       COMM    IS_NULL
------ ----- ---------- ----------
SMITH    800                     0
ALLEN   1600        300          1
WARD    1250        500          1

JONES   2975                     0
MARTIN  1250       1400          1
BLAKE   2850                     0
CLARK   2450                     0
SCOTT   3000                     0
KING    5000                     0
TURNER  1500          0          1
ADAMS   1100                     0
JAMES    950                     0
FORD    3000                     0
MILLER  1300                     0
```

By using the values returned by IS_NULL, you can easily sort NULLS first or last without interfering with the sorting of COMM.

2.6 Sorting on a Data-Dependent Key

Problem

You want to sort based on some conditional logic. For example, if JOB is SALESMAN, you want to sort on COMM; otherwise, you want to sort by SAL. You want to return the following result set:

```
ENAME             SAL JOB             COMM
---------- ---------- --------- ----------
TURNER           1500 SALESMAN          0
ALLEN            1600 SALESMAN        300
WARD             1250 SALESMAN        500
SMITH             800 CLERK
JAMES             950 CLERK
ADAMS            1100 CLERK
MILLER           1300 CLERK
MARTIN           1250 SALESMAN       1400
CLARK            2450 MANAGER
BLAKE            2850 MANAGER
JONES            2975 MANAGER
```

```
SCOTT            3000  ANALYST
FORD             3000  ANALYST
KING             5000  PRESIDENT
```

Solution

Use a CASE expression in the ORDER BY clause:

```
1 select ename,sal,job,comm
2   from emp
3  order by case when job = 'SALESMAN' then comm else sal end
```

Discussion

You can use the CASE expression to dynamically change how results are sorted. The values passed to the ORDER BY look as follows:

```
select ename,sal,job,comm,
       case when job = 'SALESMAN' then comm else sal end as ordered
  from emp
 order by 5

ENAME             SAL JOB             COMM    ORDERED
---------- ---------- --------- ---------- ----------
TURNER           1500 SALESMAN           0          0
ALLEN            1600 SALESMAN         300        300
WARD1             250 SALESMAN         500        500
SMITH             800 CLERK                       800
JAMES             950 CLERK                       950
ADAMS            1100 CLERK                      1100
MILLER           1300 CLERK                      1300
MARTIN           1250 SALESMAN        1400       1400
CLARK2            450 MANAGER                    2450
BLAKE2            850 MANAGER                    2850
JONES2            975 MANAGER                    2975
SCOTT            3000 ANALYST                    3000
FORD             3000 ANALYST                    3000
KING             5000 PRESIDENT                  5000
```

2.7 Summing Up

Sorting query results is one of the core skills for any user of SQL. The ORDER BY clause can be very powerful, but as we have seen in this chapter, still often requires some nuance to use effectively. It's important to master its use, as many of the recipes in the later chapters depend on it.

CHAPTER 3

Working with Multiple Tables

This chapter introduces the use of joins and set operations to combine data from multiple tables. Joins are the foundation of SQL. Set operations are also important. If you want to master the complex queries found in the later chapters of this book, you must start here, with joins and set operations.

3.1 Stacking One Rowset atop Another

Problem

You want to return data stored in more than one table, conceptually stacking one result set atop the other. The tables do not necessarily have a common key, but their columns do have the same data types. For example, you want to display the name and department number of the employees in department 10 in table EMP, along with the name and department number of each department in table DEPT. You want the result set to look like the following:

```
ENAME_AND_DNAME      DEPTNO
---------------  ----------
CLARK                    10
KING                     10
MILLER                   10
----------
ACCOUNTING               10
RESEARCH                 20
SALES                    30
OPERATIONS               40
```

Solution

Use the set operation UNION ALL to combine rows from multiple tables:

```
1  select ename as ename_and_dname, deptno
2    from emp
3   where deptno = 10
4   union all
5  select '----------', null
6    from t1
7   union all
8  select dname, deptno
9    from dept
```

Discussion

UNION ALL combines rows from multiple row sources into one result set. As with all set operations, the items in all the SELECT lists must match in number and data type. For example, both of the following queries will fail:

```
select deptno   |  select deptno, dname
  from dept     |    from dept
 union all      |   union all
select ename    |  select deptno
  from emp      |    from emp
```

It is important to note, UNION ALL will include duplicates if they exist. If you want to filter out duplicates, use the UNION operator. For example, a UNION between EMP.DEPTNO and DEPT.DEPTNO returns only four rows:

```
select deptno
  from emp
 union
select deptno
  from dept

   DEPTNO
---------
       10
       20
       30
       40
```

Specifying UNION rather than UNION ALL will most likely result in a sort operation to eliminate duplicates. Keep this in mind when working with large result sets. Using UNION is roughly equivalent to the following query, which applies DISTINCT to the output from a UNION ALL:

```
select distinct deptno
  from (
select deptno
  from emp
 union all
select deptno
  from dept
        )
```

```
   DEPTNO
---------
       10
       20
       30
       40
```

You wouldn't use DISTINCT in a query unless you had to, and the same rule applies for UNION: don't use it instead of UNION ALL unless you have to. For example, although in this book we have limited the number of tables for teaching purposes, in real life if you are querying one table, there may be a more suitable way to query a single table.

3.2 Combining Related Rows

Problem

You want to return rows from multiple tables by joining on a known common column or joining on columns that share common values. For example, you want to display the names of all employees in department 10 along with the location of each employee's department, but that data is stored in two separate tables. You want the result set to be the following:

```
ENAME       LOC
----------  ----------
CLARK       NEW YORK
KING        NEW YORK
MILLER      NEW YORK
```

Solution

Join table EMP to table DEPT on DEPTNO:

```
1 select e.ename, d.loc
2   from emp e, dept d
3  where e.deptno = d.deptno
4    and e.deptno = 10
```

Discussion

The solution is an example of a *join*, or more accurately an *equi-join*, which is a type of *inner join*. A join is an operation that combines rows from two tables into one. An equi-join is one in which the join condition is based on an equality condition (e.g., where one department number equals another). An inner join is the original type of join; each row returned contains data from each table.

Conceptually, the result set from a join is produced by first creating a Cartesian product (all possible combinations of rows) from the tables listed in the FROM clause, as shown here:

```
select e.ename, d.loc,
       e.deptno as emp_deptno,
       d.deptno as dept_deptno
  from emp e, dept d
 where e.deptno = 10

ENAME      LOC            EMP_DEPTNO DEPT_DEPTNO
---------- -------------  ---------- -----------
CLARK      NEW YORK               10          10
KING       NEW YORK               10          10
MILLER     NEW YORK               10          10
CLARK      DALLAS                 10          20

KING       DALLAS                 10          20
MILLER     DALLAS                 10          20
CLARK      CHICAGO                10          30
KING       CHICAGO                10          30
MILLER     CHICAGO                10          30
CLARK      BOSTON                 10          40
KING       BOSTON                 10          40
MILLER     BOSTON                 10          40
```

Every employee in table EMP (in department 10) is returned along with every department in table DEPT. Then, the expression in the WHERE clause involving e.deptno and d.deptno (the join) restricts the result set such that the only rows returned are the ones where EMP.DEPTNO and DEPT.DEPTNO are equal:

```
select e.ename, d.loc,
       e.deptno as emp_deptno,
       d.deptno as dept_deptno
  from emp e, dept d
 where e.deptno = d.deptno
   and e.deptno = 10

ENAME       LOC             EMP_DEPTNO  DEPT_DEPTNO
----------  --------------  ----------  -----------
CLARK       NEW YORK                10           10
KING        NEW YORK                10           10
MILLER      NEW YORK                10           10
```

An alternative solution makes use of an explicit JOIN clause (the INNER keyword is optional):

```
select e.ename, d.loc
  from emp e inner join dept d
    on (e.deptno = d.deptno)
 where e.deptno = 10
```

Use the JOIN clause if you prefer to have the join logic in the FROM clause rather than the WHERE clause. Both styles are ANSI compliant and work on all the latest versions of the RDBMSs in this book.

3.3 Finding Rows in Common Between Two Tables

Problem

You want to find common rows between two tables, but there are multiple columns on which you can join. For example, consider the following view V created from the EMP table for teaching purposes:

```
create view V
as
select ename,job,sal
  from emp
 where job = 'CLERK'

  select * from V

ENAME       JOB              SAL
----------  --------- ----------
SMITH       CLERK            800
ADAMS       CLERK           1100
JAMES       CLERK            950
MILLER      CLERK           1300
```

Only clerks are returned from view V. However, the view does not show all possible EMP columns. You want to return the EMPNO, ENAME, JOB, SAL, and DEPTNO of all employees in EMP that match the rows from view V. You want the result set to be the following:

```
   EMPNO  ENAME       JOB              SAL     DEPTNO
--------  ----------  --------- ---------- ---------
    7369  SMITH       CLERK            800         20
    7876  ADAMS       CLERK           1100         20
    7900  JAMES       CLERK            950         30
    7934  MILLER      CLERK           1300         10
```

Solution

Join the tables on all the columns necessary to return the correct result. Alternatively, use the set operation INTERSECT to avoid performing a join and instead return the intersection (common rows) of the two tables.

MySQL and SQL Server

Join table EMP to view V using multiple join conditions:

```
1 select e.empno,e.ename,e.job,e.sal,e.deptno
2   from emp e, V
3  where e.ename = v.ename
4    and e.job   = v.job
5    and e.sal   = v.sal
```

Alternatively, you can perform the same join via the JOIN clause:

```
1 select e.empno,e.ename,e.job,e.sal,e.deptno
2   from emp e join V
3     on (    e.ename   = v.ename
4        and e.job     = v.job
5        and e.sal     = v.sal )
```

DB2, Oracle, and PostgreSQL

The MySQL and SQL Server solution also works for DB2, Oracle, and PostgreSQL. It's the solution you should use if you need to return values from view V.

If you do not actually need to return columns from view V, you may use the set operation INTERSECT along with an IN predicate:

```
1 select empno,ename,job,sal,deptno
2   from emp
3  where (ename,job,sal) in (
4   select ename,job,sal from emp
5  intersect
6   select ename,job,sal from V
7  )
```

Discussion

When performing joins, you must consider the proper columns to join in order to return correct results. This is especially important when rows can have common values for some columns while having different values for others.

The set operation INTERSECT will return rows common to both row sources. When using INTERSECT, you are required to compare the same number of items, having the same data type, from two tables. When working with set operations, keep in mind that, by default, duplicate rows will not be returned.

3.4 Retrieving Values from One Table That Do Not Exist in Another

Problem

You want to find those values in one table, call it the source table, that do not also exist in some target table. For example, you want to find which departments (if any)

in table DEPT do not exist in table EMP. In the example data, DEPTNO 40 from table DEPT does not exist in table EMP, so the result set should be the following:

```
    DEPTNO
----------
        40
```

Solution

Having functions that perform set difference is particularly useful for this problem. DB2, PostgreSQL, SQL Server, and Oracle all support set difference operations. If your DBMS does not support a set difference function, use a subquery as shown for MySQL.

DB2, PostgreSQL, and SQL Server

Use the set operation EXCEPT:

```
1 select deptno from dept
2 except
3 select deptno from emp
```

Oracle

Use the set operation MINUS:

```
1 select deptno from dept
2 minus
3 select deptno from emp
```

MySQL

Use a subquery to return all DEPTNOs from table EMP into an outer query that searches table DEPT for rows that are not among the rows returned from the subquery:

```
1 select deptno
2   from dept
3  where deptno not in (select deptno from emp)
```

Discussion

DB2, PostgreSQL, and SQL Server

Set difference functions make this operation easy. The EXCEPT operator takes the first result set and removes from it all rows found in the second result set. The operation is very much like a subtraction.

There are restrictions on the use of set operators, including EXCEPT. Data types and number of values to compare must match in both SELECT lists. Additionally,

EXCEPT will not return duplicates and, unlike a subquery using NOT IN, NULLs do not present a problem (see the discussion for MySQL). The EXCEPT operator will return rows from the upper query (the query before the EXCEPT) that do not exist in the lower query (the query after the EXCEPT).

Oracle

The Oracle solution is identical to the solution using the EXCEPT operator; however, Oracle calls its set difference operator MINUS instead of EXCEPT. Otherwise, the preceding explanation applies to Oracle as well.

MySQL

The subquery will return all DEPTNOs from table EMP. The outer query returns all DEPTNOs from table DEPT that are "not in" or "not included in" the result set returned from the subquery.

Duplicate elimination is something you'll want to consider when using the MySQL solutions. The EXCEPT- and MINUS-based solutions used for the other platforms eliminate duplicate rows from the result set, ensuring that each DEPTNO is reported only one time. Of course, that can only be the case anyway, as DEPTNO is a key field in my example data. Were DEPTNO not a key field, you could use DISTINCT as follows to ensure that each DEPTNO value missing from EMP is reported only once:

```
select distinct deptno
  from dept
 where deptno not in (select deptno from emp)
```

Be mindful of NULLs when using NOT IN. Consider the following table, NEW_DEPT:

```
create table new_dept(deptno integer)
insert into new_deptvalues (10)
insert into new_dept values (50)
insert into new_dept values (null)
```

If you try to find the DEPTNOs in table DEPT that do not exist in table NEW_DEPT and use a subquery with NOT IN, you'll find that the query returns no rows:

```
select *
  from dept
 where deptno not in (select deptno from new_dept)
```

DEPTNOs 20, 30, and 40 are not in table NEW_DEPT, yet were not returned by the query. Why? The reason is the NULL value present in table NEW_DEPT. Three rows are returned by the subquery, with DEPTNOs of 10, 50, and NULL. IN and NOT IN are essentially OR operations and will yield different results because of how NULL values are treated by logical OR evaluations.

To understand this, examine these truth tables (Let T=true, F=false, N=null):

```
  OR | T | F | N  |
+----+---+---+----+
| T  | T | T | T  |
| F  | T | F | N  |
| N  | T | N | N  |
+----+---+---+----+

  NOT |
+-----+---+
|  T  | F |
|  F  | T |
|  N  | N |
+-----+---+

  AND | T | F | N |
+-----+---+---+---+
|  T  | T | F | N |
|  F  | F | F | F |
|  N  | N | F | N |
+-----+---+---+---+
```

Now consider the following example using IN and its equivalent using OR:

```
select deptno
  from dept
 where deptno in ( 10,50,null )

 DEPTNO
-------
     10

select deptno
  from dept
 where (deptno=10 or deptno=50 or deptno=null)

DEPTNO
-------
     10
```

Why was only DEPTNO 10 returned? There are four DEPTNOs in DEPT, (10, 20, 30, 40), and each one is evaluated against the predicate (deptno=10 or deptno=50 or deptno=null). According to the preceding truth tables, for each DEPTNO (10, 20, 30, 40), the predicate yields:

```
DEPTNO=10
(deptno=10 or deptno=50 or deptno=null)
= (10=10 or 10=50 or 10=null)
= (T or F or N)
= (T or N)
= (T)
```

```
DEPTNO=20
(deptno=10 or deptno=50 or deptno=null)
= (20=10 or 20=50 or 20=null)
= (F or F or N)
= (F or N)
= (N)

DEPTNO=30
(deptno=10 or deptno=50 or deptno=null)
= (30=10 or 30=50 or 30=null)
= (F or F or N)
= (F or N)
= (N)

DEPTNO=40
(deptno=10 or deptno=50 or deptno=null)
= (40=10 or 40=50 or 40=null)
= (F or F or N)
= (F or N)
= (N)
```

Now it is obvious why only DEPTNO 10 was returned when using IN and OR. Next, consider the same example using NOT IN and NOT OR:

```
select deptno
  from dept
 where deptno not in ( 10,50,null )

( no rows )

select deptno
  from dept
 where not (deptno=10 or deptno=50 or deptno=null)

( no rows )
```

Why are no rows returned? Let's check the truth tables:

```
DEPTNO=10
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (10=10 or 10=50 or 10=null)
= NOT (T or F or N)
= NOT (T or N)
= NOT (T)
= (F)

DEPTNO=20
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (20=10 or 20=50 or 20=null)
= NOT (F or F or N)
= NOT (F or N)
= NOT (N)
= (N)
```

```
DEPTNO=30
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (30=10 or 30=50 or 30=null)
= NOT (F or F or N)
= NOT (F or N)
= NOT (N)
= (N)

DEPTNO=40
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (40=10 or 40=50 or 40=null)
= NOT (F or F or N)
= NOT (F or N)
= NOT (N)
= (N)
```

In SQL, "TRUE or NULL" is TRUE, but "FALSE or NULL" is NULL! You must keep this in mind when using IN predicates, and when performing logical OR evaluations and NULL values are involved.

To avoid the problem with NOT IN and NULLs, use a correlated subquery in conjunction with NOT EXISTS. The term *correlated subquery* is used because rows from the outer query are referenced in the subquery. The following example is an alternative solution that will not be affected by NULL rows (going back to the original query from the "Problem" section):

```
select d.deptno
  from dept d
 where not exists (
   select 1
     from emp e
    where d.deptno = e.deptno
)

DEPTNO
----------
40

select d.deptno
  from dept d
 where not exists (
   select 1
     from new_dept nd
    where d.deptno = nd.deptno
)

DEPTNO
----------
30
40
20
```

Conceptually, the outer query in this solution considers each row in the DEPT table. For each DEPT row, the following happens:

1. The subquery is executed to see whether the department number exists in the EMP table. Note the condition D.DEPTNO = E.DEPTNO, which brings together the department numbers from the two tables.
2. If the subquery returns results, then EXISTS (…) evaluates to true and NOT EXISTS (…) thus evaluates to FALSE, and the row being considered by the outer query is discarded.
3. If the subquery returns no results, then NOT EXISTS (…) evaluates to TRUE, and the row being considered by the outer query is returned (because it is for a department not represented in the EMP table).

The items in the SELECT list of the subquery are unimportant when using a correlated subquery with EXISTS/NOT EXISTS, which is why we chose to select NULL, to force you to focus on the join in the subquery rather than the items in the SELECT list.

3.5 Retrieving Rows from One Table That Do Not Correspond to Rows in Another

Problem

You want to find rows that are in one table that do not have a match in another table, for two tables that have common keys. For example, you want to find which departments have no employees. The result set should be the following:

```
    DEPTNO  DNAME          LOC
----------  -------------  -------------
        40  OPERATIONS     BOSTON
```

Finding the department each employee works in requires an equi-join on DEPTNO from EMP to DEPT. The DEPTNO column represents the common value between tables. Unfortunately, an equi-join will not show you which department has no employees. That's because by equi-joining EMP and DEPT you are returning all rows that satisfy the join condition. Instead, you want only those rows from DEPT that do not satisfy the join condition.

This is a subtly different problem than in the preceding recipe, though at first glance they may seem the same. The difference is that the preceding recipe yields only a list of department numbers not represented in table EMP. Using this recipe, however, you can easily return other columns from the DEPT table; you can return more than just department numbers.

Solution

Return all rows from one table along with rows from another that may or may not have a match on the common column. Then, keep only those rows with no match.

DB2, MySQL, PostgreSQL, and SQL Server

Use an outer join and filter for NULLs (keyword OUTER is optional):

```
1 select d.*
2   from dept d left outer join emp e
3     on (d.deptno = e.deptno)
4  where e.deptno is null
```

Discussion

This solution works by outer joining and then keeping only rows that have no match. This sort of operation is sometimes called an *anti-join*. To get a better idea of how an anti-join works, first examine the result set without filtering for NULLs:

```
select e.ename, e.deptno as emp_deptno, d.*
  from dept d left join emp e
    on (d.deptno = e.deptno)

ENAME       EMP_DEPTNO      DEPTNO DNAME          LOC
----------  ----------  ---------- -------------- -------------
SMITH               20          20 RESEARCH       DALLAS
ALLEN               30          30 SALES          CHICAGO
WARD                30          30 SALES          CHICAGO
JONES               20          20 RESEARCH       DALLAS
MARTIN              30          30 SALES          CHICAGO
BLAKE               30          30 SALES          CHICAGO
CLARK               10          10 ACCOUNTING     NEW YORK
SCOTT               20          20 RESEARCH       DALLAS
KING                10          10 ACCOUNTING     NEW YORK
TURNER              30          30 SALES          CHICAGO
ADAMS               20          20 RESEARCH       DALLAS
JAMES               30          30 SALES          CHICAGO
FORD                20          20 RESEARCH       DALLAS
MILLER              10          10 ACCOUNTING     NEW YORK
                                40 OPERATIONS     BOSTON
```

Notice, the last row has a NULL value for EMP.ENAME and EMP_DEPTNO. That's because no employees work in department 40. The solution uses the WHERE clause to keep only rows where EMP_DEPTNO is NULL (thus keeping only rows from DEPT that have no match in EMP).

3.6 Adding Joins to a Query Without Interfering with Other Joins

Problem

You have a query that returns the results you want. You need additional information, but when trying to get it, you lose data from the original result set. For example, you want to return all employees, the location of the department in which they work, and the date they received a bonus. For this problem, the EMP_BONUS table contains the following data:

```
select * from emp_bonus

     EMPNO  RECEIVED          TYPE
----------  ----------- ----------
      7369  14-MAR-2005          1
      7900  14-MAR-2005          2
      7788  14-MAR-2005          3
```

The query you start with looks like this:

```
select e.ename, d.loc
  from emp e, dept d
 where e.deptno=d.deptno

  ENAME      LOC
  ---------- -------------
  SMITH      DALLAS
  ALLEN      CHICAGO
  WARD       CHICAGO
  JONES      DALLAS
  MARTIN     CHICAGO
  BLAKE      CHICAGO
  CLARK      NEW YORK
  SCOTT      DALLAS
  KING       NEW YORK
  TURNER     CHICAGO
  ADAMS      DALLAS
  JAMES      CHICAGO
  FORD       DALLAS
  MILLER     NEW YORK
```

You want to add to these results the date a bonus was given to an employee, but joining to the EMP_BONUS table returns fewer rows than you want because not every employee has a bonus:

```
select e.ename, d.loc,eb.received
  from emp e, dept d, emp_bonus eb
 where e.deptno=d.deptno
   and e.empno=eb.empno
```

```
ENAME       LOC           RECEIVED
----------  ------------- -----------
SCOTT       DALLAS        14-MAR-2005
SMITH       DALLAS        14-MAR-2005
JAMES       CHICAGO       14-MAR-2005
```

Your desired result set is the following:

```
ENAME       LOC            RECEIVED
----------  -------------  -----------
ALLEN       CHICAGO
WARD        CHICAGO
MARTIN      CHICAGO
JAMES       CHICAGO        14-MAR-2005
TURNER      CHICAGO
BLAKE       CHICAGO
SMITH       DALLAS         14-MAR-2005
FORD        DALLAS
ADAMS       DALLAS
JONES       DALLAS
SCOTT       DALLAS         14-MAR-2005
CLARK       NEW YORK
KING        NEW YORK
MILLER      NEW YORK
```

Solution

You can use an outer join to obtain the additional information without losing the data from the original query. First join table EMP to table DEPT to get all employees and the location of the department they work, then outer join to table EMP_ BONUS to return the date of the bonus if there is one. The following is the DB2, MySQL, PostgreSQL, and SQL server syntax:

```
1 select e.ename, d.loc, eb.received
2   from emp e join dept d
3     on (e.deptno=d.deptno)
4   left join emp_bonus eb
5     on (e.empno=eb.empno)
6  order by 2
```

You can also use a scalar subquery (a subquery placed in the SELECT list) to mimic an outer join:

```
1 select e.ename, d.loc,
2        (select eb.received from emp_bonus eb
3          where eb.empno=e.empno) as received
4   from emp e, dept d
5  where e.deptno=d.deptno
6  order by 2
```

The scalar subquery solution will work across all platforms.

Discussion

An outer join will return all rows from one table and matching rows from another. See the previous recipe for another example of such a join. The reason an outer join works to solve this problem is that it does not result in any rows being eliminated that would otherwise be returned. The query will return all the rows it would return without the outer join. And it also returns the received date, if one exists.

Use of a scalar subquery is also a convenient technique for this sort of problem, as it does not require you to modify already correct joins in your main query. Using a scalar subquery is an easy way to tack on extra data to a query without compromising the current result set. When working with scalar subqueries, you must ensure they return a scalar (single) value. If a subquery in the SELECT list returns more than one row, you will receive an error.

See Also

See Recipe 14.10 for a workaround to the problem of not being able to return multiple rows from a SELECT-list subquery.

3.7 Determining Whether Two Tables Have the Same Data

Problem

You want to know whether two tables or views have the same data (cardinality and values). Consider the following view:

```
create view V
as
select * from emp where deptno != 10
 union all
select * from emp where ename = 'WARD'

select * from V

EMPNO ENAME       JOB        MGR  HIREDATE      SAL  COMM DEPTNO
----- ----------  --------- ----- ----------- ----- ----- ------
 7369 SMITH       CLERK      7902 17-DEC-2005   800           20
 7499 ALLEN       SALESMAN   7698 20-FEB-2006  1600   300     30
 7521 WARD        SALESMAN   7698 22-FEB-2006  1250   500     30
 7566 JONES       MANAGER    7839 02-APR-2006  2975           20
 7654 MARTIN      SALESMAN   7698 28-SEP-2006  1250  1400     30
 7698 BLAKE       MANAGER    7839 01-MAY-2006  2850           30
 7788 SCOTT       ANALYST    7566 09-DEC-2007  3000           20
 7844 TURNER      SALESMAN   7698 08-SEP-2006  1500     0     30
 7876 ADAMS       CLERK      7788 12-JAN-2008  1100           20
 7900 JAMES       CLERK      7698 03-DEC-2006   950           30
```

```
 7902 FORD        ANALYST    7566 03-DEC-2006  3000           20
 7521 WARD        SALESMAN   7698 22-FEB-2006  1250   500     30
```

You want to determine whether this view has exactly the same data as table EMP. The row for employee WARD is duplicated to show that the solution will reveal not only different data but duplicates as well. Based on the rows in table EMP, the difference will be the three rows for employees in department 10 and the two rows for employee WARD. You want to return the following result set:

```
EMPNO ENAME       JOB         MGR  HIREDATE      SAL   COMM DEPTNO CNT
----- ----------  ---------  ----- ----------- -----  ----- ------ ---
 7521 WARD        SALESMAN    7698 22-FEB-2006  1250    500     30   1
 7521 WARD        SALESMAN    7698 22-FEB-2006  1250    500     30   2
 7782 CLARK       MANAGER     7839 09-JUN-2006  2450            10   1
 7839 KING        PRESIDENT        17-NOV-2006  5000            10   1
 7934 MILLER      CLERK       7782 23-JAN-2007  1300            10   1
```

Solution

Functions that perform SET difference MINUS or EXCEPT, depending on your DBMS, make the problem of comparing tables a relatively easy one to solve. If your DBMS does not offer such functions, you can use a correlated subquery.

DB2 and PostgreSQL

Use the set operations EXCEPT and UNION ALL to find the difference between view V and table EMP combined with the difference between table EMP and view V:

```
 1  (
 2   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
 3          count(*) as cnt
 4     from V
 5    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
 6   except
 7   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
 8          count(*) as cnt
 9     from emp
10    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
11   )
12    union all
13   (
14   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
15          count(*) as cnt
16     from emp
17    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
18   except
19   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
20          count(*) as cnt
21     from v
22    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
23  )
```

Oracle

Use the set operations MINUS and UNION ALL to find the difference between view V and table EMP combined with the difference between table EMP and view V:

```
(
 select empno,ename,job,mgr,hiredate,sal,comm,deptno,
        count(*) as cnt
   from  V
  group by empno,ename,job,mgr,hiredate,sal,comm,deptno
  minus
 select empno,ename,job,mgr,hiredate,sal,comm,deptno,
        count(*) as cnt
   from emp
  group by empno,ename,job,mgr,hiredate,sal,comm,deptno
 )
   union all
 (
 select empno,ename,job,mgr,hiredate,sal,comm,deptno,
        count(*) as cnt
   from emp
  group by empno,ename,job,mgr,hiredate,sal,comm,deptno
  minus
 select empno,ename,job,mgr,hiredate,sal,comm,deptno,
        count(*) as cnt
   from v
  group by empno,ename,job,mgr,hiredate,sal,comm,deptno
)
```

MySQL and SQL Server

Use a correlated subquery and UNION ALL to find the rows in view V and not in table EMP combined with the rows in table EMP and not in view V:

```
select *
  from (
select e.empno,e.ename,e.job,e.mgr,e.hiredate,
       e.sal,e.comm,e.deptno, count(*) as cnt
  from emp e
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
       ) e
 where not exists (
select null
  from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
       v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
       ) v
  where v.empno     = e.empno
     and v.ename    = e.ename
```

```
20       and v.job      = e.job
21       and coalesce(v.mgr,0) = coalesce(e.mgr,0)
22       and v.hiredate = e.hiredate
23       and v.sal      = e.sal
24       and v.deptno   = e.deptno
25       and v.cnt      = e.cnt
26       and coalesce(v.comm,0) = coalesce(e.comm,0)
27  )
28     union all
29    select *
30      from (
31    select v.empno,v.ename,v.job,v.mgr,v.hiredate,
32           v.sal,v.comm,v.deptno, count(*) as cnt
33      from v
34    group by empno,ename,job,mgr,hiredate,
35             sal,comm,deptno
36          ) v
37    where not exists (
38   select null
39     from (
40   select e.empno,e.ename,e.job,e.mgr,e.hiredate,
41          e.sal,e.comm,e.deptno, count(*) as cnt
42     from emp e
43    group by empno,ename,job,mgr,hiredate,
44             sal,comm,deptno
45          ) e
46    where v.empno     = e.empno
47      and v.ename     = e.ename
48      and v.job       = e.job
49      and coalesce(v.mgr,0) = coalesce(e.mgr,0)
50      and v.hiredate  = e.hiredate
51      and v.sal       = e.sal
52      and v.deptno    = e.deptno
53      and v.cnt       = e.cnt
54      and coalesce(v.comm,0) = coalesce(e.comm,0)
55  )
```

Discussion

Despite using different techniques, the concept is the same for all solutions:

1. Find rows in table EMP that do not exist in view V.
2. Combine (UNION ALL) those rows with rows from view V that do not exist in table EMP.

If the tables in question are equal, then no rows are returned. If the tables are different, the rows causing the difference are returned. As an easy first step when comparing tables, you can compare the cardinalities alone rather than including them with the data comparison.

The following query is a simple example of this and will work on all DBMSs:

```
select count(*)
  from emp
 union
select count(*)
  from dept

COUNT(*)
--------
       4
      14
```

Because UNION will filter out duplicates, only one row will be returned if the tables' cardinalities are the same. Because two rows are returned in this example, you know that the tables do not contain identical rowsets.

DB2, Oracle, and PostgreSQL

MINUS and EXCEPT work in the same way, so we will use EXCEPT for this discussion. The queries before and after the UNION ALL are similar. So, to understand how the solution works, simply execute the query prior to the UNION ALL by itself. The following result set is produced by executing lines 1–11 in the "Solution" section:

```
(
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from V
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
  except
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from emp
  group by empno,ename,job,mgr,hiredate,sal,comm,deptno
)

 EMPNO ENAME      JOB        MGR  HIREDATE      SAL  COMM DEPTNO CNT
 ----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521  WARD       SALESMAN   7698 22-FEB-2006  1250   500     30   2
```

The result set represents a row found in view V that is either not in table EMP, or has a different cardinality than that same row in table EMP. In this case, the duplicate row for employee WARD is found and returned. If you're still having trouble understanding how the result set is produced, run each query on either side of EXCEPT individually. You'll notice the only difference between the two result sets is the CNT for employee WARD returned by view V.

The portion of the query after the UNION ALL does the opposite of the query preceding UNION ALL. The query returns rows in table EMP not in view V:

```
(
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from emp
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
   minus
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from v
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
)

EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
7521  WARD       SALESMAN   7698 22-FEB-2006  1250   500     30   1
7782  CLARK      MANAGER    7839 09-JUN-2006  2450           10   1
7839  KING       PRESIDENT       17-NOV-2006  5000           10   1
7934  MILLER     CLERK      7782 23-JAN-2007  1300           10   1
```

The results are then combined by UNION ALL to produce the final result set.

MySQL and SQL Server

The queries before and after the UNION ALL are similar. To understand how the subquery-based solution works, simply execute the query prior to the UNION ALL by itself. The following query is from lines 1–27 in the solution:

```
select *
   from (
 select e.empno,e.ename,e.job,e.mgr,e.hiredate,
        e.sal,e.comm,e.deptno, count(*) as cnt
   from emp e
  group by empno,ename,job,mgr,hiredate,
           sal,comm,deptno
        ) e
  where not exists (
select null
  from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
       v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
       ) v
  where v.empno    = e.empno
    and v.ename    = e.ename
    and v.job      = e.job
    and v.mgr      = e.mgr
    and v.hiredate = e.hiredate
```

```
    and v.sal      = e.sal
    and v.deptno   = e.deptno
    and v.cnt      = e.cnt
    and coalesce(v.comm,0) = coalesce(e.comm,0)
 )

EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-2006  1250   500     30   1
 7782 CLARK      MANAGER    7839 09-JUN-2006  2450           10   1
 7839 KING       PRESIDENT       17-NOV-2006  5000           10   1
 7934 MILLER     CLERK      7782 23-JAN-2007  1300           10   1
```

Notice that the comparison is not between table EMP and view V, but rather between inline view E and inline view V. The cardinality for each row is found and returned as an attribute for that row. You are comparing each row and its occurrence count. If you are having trouble understanding how the comparison works, run the subqueries independently. The next step is to find all rows (including CNT) in inline view E that do not exist in inline view V. The comparison uses a correlated subquery and NOT EXISTS. The joins will determine which rows are the same, and the result will be all rows from inline view E that are not the rows returned by the join. The query after the UNION ALL does the opposite; it finds all rows in inline view V that do not exist in inline view E:

```
select *
  from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
       v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
       ) v
  where not exists (
select null
  from (
 select e.empno,e.ename,e.job,e.mgr,e.hiredate,
        e.sal,e.comm,e.deptno, count(*) as cnt
   from emp e
  group by empno,ename,job,mgr,hiredate,
           sal,comm,deptno
       ) e
  where v.empno    = e.empno
    and v.ename    = e.ename
    and v.job      = e.job
    and v.mgr      = e.mgr
    and v.hiredate = e.hiredate
    and v.sal      = e.sal
    and v.deptno   = e.deptno
    and v.cnt      = e.cnt
    and coalesce(v.comm,0) = coalesce(e.comm,0)
)
```

```
EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-2006  1250   500     30   2
```

The results are then combined by UNION ALL to produce the final result set.

Ales Spetic and Jonathan Gennick give an alternate solution in their book *Transact-SQL Cookbook* (O'Reilly). See the section "Comparing Two Sets for Equality" in Chapter 2 of their book.

3.8 Identifying and Avoiding Cartesian Products

Problem

You want to return the name of each employee in department 10 along with the location of the department. The following query is returning incorrect data:

```
select e.ename, d.loc
  from emp e, dept d
 where e.deptno = 10

ENAME       LOC
----------  -------------
CLARK       NEW YORK
CLARK       DALLAS
CLARK       CHICAGO
CLARK       BOSTON
KING        NEW YORK
KING        DALLAS
KING        CHICAGO
KING        BOSTON
MILLER      NEW YORK
MILLER      DALLAS
MILLER      CHICAGO
MILLER      BOSTON
```

The correct result set is the following:

```
ENAME       LOC
----------  ---------
CLARK       NEW YORK
KING        NEW YORK
MILLER      NEW YORK
```

Solution

Use a join between the tables in the FROM clause to return the correct result set:

```
1 select e.ename, d.loc
2   from emp e, dept d
3  where e.deptno = 10
4    and d.deptno = e.deptno
```

Discussion

Let's look at the data in the DEPT table:

```
select * from dept

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

You can see that department 10 is in New York, and thus you can know that returning employees with any location other than New York is incorrect. The number of rows returned by the incorrect query is the product of the cardinalities of the two tables in the FROM clause. In the original query, the filter on EMP for department 10 will result in three rows. Because there is no filter for DEPT, all four rows from DEPT are returned. Three multiplied by four is twelve, so the incorrect query returns twelve rows. Generally, to avoid a Cartesian product, you would apply the n–1 rule where n represents the number of tables in the FROM clause and n–1 represents the minimum number of joins necessary to avoid a Cartesian product. Depending on what the keys and join columns in your tables are, you may very well need more than n–1 joins, but n–1 is a good place to start when writing queries.

When used properly, Cartesian products can be useful. Common uses of Cartesian products include transposing or pivoting (and unpivoting) a result set, generating a sequence of values, and mimicking a loop (although the last two may also be accomplished using a recursive CTE).

3.9 Performing Joins When Using Aggregates

Problem

You want to perform an aggregation, but your query involves multiple tables. You want to ensure that joins do not disrupt the aggregation. For example, you want to

find the sum of the salaries for employees in department 10 along with the sum of their bonuses. Some employees have more than one bonus, and the join between table EMP and table EMP_BONUS is causing incorrect values to be returned by the aggregate function SUM. For this problem, table EMP_BONUS contains the following data:

```
select * from emp_bonus

EMPNO RECEIVED          TYPE
----- ----------- ----------
 7934 17-MAR-2005          1
 7934 15-FEB-2005          2
 7839 15-FEB-2005          3
 7782 15-FEB-2005          1
```

Now, consider the following query that returns the salary and bonus for all employees in department 10. Table BONUS.TYPE determines the amount of the bonus. A type 1 bonus is 10% of an employee's salary, type 2 is 20%, and type 3 is 30%.

```
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3
             end as bonus
 from emp e, emp_bonus eb
where e.empno  = eb.empno
  and e.deptno = 10


  EMPNO ENAME             SAL     DEPTNO     BONUS
------- ---------- ---------- ---------- ---------
   7934 MILLER           1300         10       130
   7934 MILLER           1300         10       260
   7839 KING             5000         10      1500
   7782 CLARK            2450         10       245
```

So far, so good. However, things go awry when you attempt a join to the EMP_BONUS table to sum the bonus amounts:

```
select deptno,
       sum(sal) as total_sal,
       sum(bonus) as total_bonus
  from (
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
```

```
                 else .3
             end as bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10
       ) x
 group by deptno


DEPTNO   TOTAL_SAL  TOTAL_BONUS
------ -----------  -----------
    10       10050         2135
```

While the TOTAL_BONUS is correct, the TOTAL_SAL is incorrect. The sum of all salaries in department 10 is 8750, as the following query shows:

```
select sum(sal) from emp where deptno=10

  SUM(SAL)
----------
      8750
```

Why is TOTAL_SAL incorrect? The reason is the duplicate rows in the SAL column created by the join. Consider the following query, which joins tables EMP and EMP_BONUS:

```
select e.ename,
       e.sal
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10

ENAME             SAL
---------- ----------
CLARK            2450
KING             5000
MILLER           1300
MILLER           1300
```

Now it is easy to see why the value for TOTAL_SAL is incorrect: MILLER's salary is counted twice. The final result set that you are really after is:

```
DEPTNO TOTAL_SAL TOTAL_BONUS
------ --------- -----------
    10      8750        2135
```

Solution

You have to be careful when computing aggregates across joins. Typically when duplicates are returned due to a join, you can avoid miscalculations by aggregate functions in two ways: you can simply use the keyword DISTINCT in the call to the aggregate function, so only unique instances of each value are used in the computation; or you

can perform the aggregation first (in an inline view) prior to joining, thus avoiding the incorrect computation by the aggregate function because the aggregate will already be computed before you even join, thus avoiding the problem altogether. The solutions that follow use DISTINCT. The "Discussion" section will discuss the technique of using an inline view to perform the aggregation prior to joining.

MySQL and PostgreSQL

Perform a sum of only the DISTINCT salaries:

```
select deptno,
       sum(distinct sal) as total_sal,
       sum(bonus) as total_bonus
  from (
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3
              end as bonus
  from emp e, emp_bonus eb
  where e.empno = eb.empno
   and e.deptno = 10
       ) x
 group by deptno
```

DB2, Oracle, and SQL Server

These platforms support the preceding solution, but they also support an alternative solution using the window function SUM OVER:

```
select distinct deptno,total_sal,total_bonus
  from (
select e.empno,
       e.ename,
       sum(distinct e.sal) over
       (partition by e.deptno) as total_sal,
        e.deptno,
        sum(e.sal*case when eb.type = 1 then .1
                       when eb.type = 2 then .2
                       else .3 end) over
        (partition by deptno) as total_bonus
   from emp e, emp_bonus eb
  where e.empno = eb.empno
    and e.deptno = 10
        ) x
```

Discussion

MySQL and PostgreSQL

The second query in the "Problem" section of this recipe joins table EMP and table EMP_BONUS and returns two rows for employee MILLER, which is what causes the error on the sum of EMP.SAL (the salary is added twice). The solution is to simply sum the distinct EMP.SAL values that are returned by the query. The following query is an alternative solution—necessary if there could be duplicate values in the column you are summing. The sum of all salaries in department 10 is computed first, and that row is then joined to table EMP, which is then joined to table EMP_BONUS.

The following query works for all DBMSs:

```
select d.deptno,
       d.total_sal,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end) as total_bonus
  from emp e,
       emp_bonus eb,
       (
select deptno, sum(sal) as total_sal
  from emp
 where deptno = 10
 group by deptno
        ) d
 where e.deptno = d.deptno
   and e.empno = eb.empno
 group by d.deptno,d.total_sal


   DEPTNO  TOTAL_SAL  TOTAL_BONUS
--------- ---------- ------------
       10       8750         2135
```

DB2, Oracle, and SQL Server

This alternative solution takes advantage of the window function SUM OVER. The following query is taken from lines 3–14 in "Solution" and returns the following result set:

```
select e.empno,
          e.ename,
          sum(distinct e.sal) over
          (partition by e.deptno) as total_sal,
          e.deptno,
          sum(e.sal*case when eb.type = 1 then .1
                         when eb.type = 2 then .2
                        else .3 end) over
         (partition by deptno) as total_bonus
```

```
    from emp e, emp_bonus eb
   where e.empno  = eb.empno
     and e.deptno = 10

EMPNO ENAME        TOTAL_SAL   DEPTNO  TOTAL_BONUS
----- ----------  ----------   ------  -----------
 7934 MILLER            8750       10         2135
 7934 MILLER            8750       10         2135
 7782 CLARK             8750       10         2135
 7839 KING              8750       10         2135
```

The windowing function, SUM OVER, is called twice, first to compute the sum of the distinct salaries for the defined partition or group. In this case, the partition is DEPTNO 10, and the sum of the distinct salaries for DEPTNO 10 is 8750. The next call to SUM OVER computes the sum of the bonuses for the same defined partition. The final result set is produced by taking the distinct values for TOTAL_SAL, DEPTNO, and TOTAL_BONUS.

3.10 Performing Outer Joins When Using Aggregates

Problem

Begin with the same problem as in Recipe 3.9, but modify table EMP_BONUS such that the difference in this case is not all employees in department 10 have been given bonuses. Consider the EMP_BONUS table and a query to (ostensibly) find both the sum of all salaries for department 10 and the sum of all bonuses for all employees in department 10:

```
select * from emp_bonus

     EMPNO RECEIVED          TYPE
---------- ----------- ----------
      7934 17-MAR-2005          1
      7934 15-FEB-2005          2


select deptno,
       sum(sal) as total_sal,
       sum(bonus) as total_bonus
  from (
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3 end as bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
```

```
  and e.deptno = 10
      )
group by deptno

DEPTNO  TOTAL_SAL TOTAL_BONUS
------ ---------- -----------
    10       2600         390
```

The result for TOTAL_BONUS is correct, but the value returned for TOTAL_SAL does not represent the sum of all salaries in department 10. The following query shows why the TOTAL_SAL is incorrect:

```
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
             else .3 end as bonus
  from emp e, emp_bonus eb
 where e.empno = eb.empno
   and e.deptno = 10

    EMPNO ENAME          SAL     DEPTNO      BONUS
--------- ---------  ------- ---------- ----------
     7934 MILLER        1300         10        130
     7934 MILLER        1300         10        260
```

Rather than sum all salaries in department 10, only the salary for MILLER is summed, and it is erroneously summed twice. Ultimately, you would like to return the following result set:

```
DEPTNO TOTAL_SAL TOTAL_BONUS
------ --------- -----------
    10      8750         390
```

Solution

The solution is similar to that of Recipe 3.9, but here you outer join to EMP_BONUS to ensure all employees from department 10 are included.

DB2, MySQL, PostgreSQL, and SQL Server

Outer join to EMP_BONUS, then perform the sum on only distinct salaries from department 10:

```
1 select deptno,
2        sum(distinct sal) as total_sal,
3        sum(bonus) as total_bonus
4   from (
5 select e.empno,
6        e.ename,
```

```
       e.sal,
       e.deptno,
       e.sal*case when eb.type is null then 0
                  when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3 end as bonus
  from emp e left outer join emp_bonus eb
    on (e.empno = eb.empno)
 where e.deptno = 10
       )
 group by deptno
```

You can also use the window function SUM OVER:

```
select distinct deptno,total_sal,total_bonus
  from (
select e.empno,
       e.ename,
       sum(distinct e.sal) over
       (partition by e.deptno) as total_sal,
       e.deptno,
       sum(e.sal*case when eb.type is null then 0
                      when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3
                 end) over
       (partition by deptno) as total_bonus
  from emp e left outer join emp_bonus eb
    on (e.empno = eb.empno)
 where e.deptno = 10
       ) x
```

Discussion

The second query in the "Problem" section of this recipe joins table EMP and table EMP_BONUS and returns only rows for employee MILLER, which is what causes the error on the sum of EMP.SAL (the other employees in DEPTNO 10 do not have bonuses, and their salaries are not included in the sum). The solution is to outer join table EMP to table EMP_BONUS so even employees without a bonus will be included in the result. If an employee does not have a bonus, NULL will be returned for EMP_BONUS.TYPE. It is important to keep this in mind as the CASE statement has been modified and is slightly different from Recipe 3.9. If EMP_BONUS.TYPE is NULL, the CASE expression returns zero, which has no effect on the sum.

The following query is an alternative solution. The sum of all salaries in department 10 is computed first, then joined to table EMP, which is then joined to table EMP_BONUS (thus avoiding the outer join). The following query works for all DBMSs:

```
select d.deptno,
       d.total_sal,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end) as total_bonus
  from emp e,
       emp_bonus eb,
      (
select deptno, sum(sal) as total_sal
  from emp
 where deptno = 10
 group by deptno
       ) d
 where e.deptno = d.deptno
   and e.empno = eb.empno
 group by d.deptno,d.total_sal


   DEPTNO  TOTAL_SAL TOTAL_BONUS
--------- ---------- -----------
       10       8750         390
```

3.11 Returning Missing Data from Multiple Tables

Problem

You want to return missing data from multiple tables simultaneously. Returning rows from table DEPT that do not exist in table EMP (any departments that have no employees) requires an outer join. Consider the following query, which returns all DEPTNOs and DNAMEs from DEPT along with the names of all the employees in each department (if there is an employee in a particular department):

```
select d.deptno,d.dname,e.ename
  from dept d left outer join emp e
    on (d.deptno=e.deptno)

   DEPTNO DNAME          ENAME
--------- -------------- ----------
       20 RESEARCH       SMITH
       30 SALES          ALLEN
       30 SALES          WARD
       20 RESEARCH       JONES
       30 SALES          MARTIN
       30 SALES          BLAKE
       10 ACCOUNTING     CLARK
       20 RESEARCH       SCOTT
       10 ACCOUNTING     KING
       30 SALES          TURNER
       20 RESEARCH       ADAMS
       30 SALES          JAMES
```

```
        20 RESEARCH       FORD
        10 ACCOUNTING     MILLER
        40 OPERATIONS
```

The last row, the OPERATIONS department, is returned despite that department not having any employees, because table EMP was outer joined to table DEPT. Now, suppose there was an employee without a department. How would you return the previous result set along with a row for the employee having no department? In other words, you want to outer join to both table EMP and table DEPT, and in the same query. After creating the new employee, a first attempt may look like this:

```
insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
select 1111,'YODA','JEDI',null,hiredate,sal,comm,null
  from emp
 where ename = 'KING'

select d.deptno,d.dname,e.ename
  from dept d right outer join emp e
    on (d.deptno=e.deptno)

    DEPTNO DNAME        ENAME
---------- ------------ ----------
        10 ACCOUNTING   MILLER
        10 ACCOUNTING   KING
        10 ACCOUNTING   CLARK
        20 RESEARCH     FORD
        20 RESEARCH     ADAMS
        20 RESEARCH     SCOTT
        20 RESEARCH     JONES
        20 RESEARCH     SMITH
        30 SALES        JAMES
        30 SALES        TURNER
        30 SALES        BLAKE
        30 SALES        MARTIN
        30 SALES        WARD
        30 SALES        ALLEN
                        YODA
```

This outer join manages to return the new employee but lost the OPERATIONS department from the original result set. The final result set should return a row for YODA as well as OPERATIONS, such as the following:

```
    DEPTNO DNAME        ENAME
---------- ------------ --------
        10 ACCOUNTING   CLARK
        10 ACCOUNTING   KING
        10 ACCOUNTING   MILLER
        20 RESEARCH     ADAMS
        20 RESEARCH     FORD
        20 RESEARCH     JONES
        20 RESEARCH     SCOTT
        20 RESEARCH     SMITH
```

```
30 SALES        ALLEN
30 SALES        BLAKE
30 SALES        JAMES
30 SALES        MARTIN
30 SALES        TURNER
30 SALES        WARD
40 OPERATIONS
                YODA
```

Solution

Use a full outer join to return missing data from both tables based on a common value.

DB2, MySQL, PostgreSQL, and SQL Server

Use the explicit FULL OUTER JOIN command to return missing rows from both tables along with matching rows:

```
1 select d.deptno,d.dname,e.ename
2   from dept d full outer join emp e
3     on (d.deptno=e.deptno)
```

Alternatively, since MySQL does not yet have a FULL OUTER JOIN, UNION the results of the two different outer joins:

```
1 select d.deptno,d.dname,e.ename
2   from dept d right outer join emp e
3     on (d.deptno=e.deptno)
4  union
5 select d.deptno,d.dname,e.ename
6   from dept d left outer join emp e
7     on (d.deptno=e.deptno)
```

Oracle

Oracle users can still use either of the preceding solutions. Alternatively, you can use Oracle's proprietary outer join syntax:

```
1 select d.deptno,d.dname,e.ename
2   from dept d, emp e
3  where d.deptno = e.deptno(+)
4  union
5 select d.deptno,d.dname,e.ename
6   from dept d, emp e
7  where d.deptno(+) = e.deptno
```

Discussion

The full outer join is simply the combination of outer joins on both tables. To see how a full outer join works "under the covers," simply run each outer join, then union the

results. The following query returns rows from table DEPT and any matching rows from table EMP (if any):

```
select d.deptno,d.dname,e.ename
  from dept d left outer join emp e
    on (d.deptno = e.deptno)

  DEPTNO DNAME          ENAME
  ------ -------------- ----------
      20 RESEARCH       SMITH
      30 SALES          ALLEN
      30 SALES          WARD
      20 RESEARCH       JONES
      30 SALES          MARTIN
      30 SALES          BLAKE
      10 ACCOUNTING     CLARK
      20 RESEARCH       SCOTT
      10 ACCOUNTING     KING
      30 SALES          TURNER
      20 RESEARCH       ADAMS
      30 SALES          JAMES
      20 RESEARCH       FORD
      10 ACCOUNTING     MILLER
      40 OPERATIONS
```

This next query returns rows from table EMP and any matching rows from table DEPT (if any):

```
select d.deptno,d.dname,e.ename
  from dept d right outer join emp e
    on (d.deptno = e.deptno)

 DEPTNO DNAME          ENAME
 ------ -------------- ----------
     10 ACCOUNTING     MILLER
     10 ACCOUNTING     KING
     10 ACCOUNTING     CLARK
     20 RESEARCH       FORD
     20 RESEARCH       ADAMS
     20 RESEARCH       SCOTT
     20 RESEARCH       JONES
     20 RESEARCH       SMITH
     30 SALES          JAMES
     30 SALES          TURNER
     30 SALES          BLAKE
     30 SALES          MARTIN
     30 SALES          WARD
     30 SALES          ALLEN
                       YODA
```

The results from these two queries are unioned to provide the final result set.

3.12 Using NULLs in Operations and Comparisons

Problem

NULL is never equal to or not equal to any value, not even itself, but you want to evaluate values returned by a nullable column like you would evaluate real values. For example, you want to find all employees in EMP whose commission (COMM) is less than the commission of employee WARD. Employees with a NULL commission should be included as well.

Solution

Use a function such as COALESCE to transform the NULL value into a real value that can be used in standard evaluation:

```
1 select ename,comm
2   from emp
3  where coalesce(comm,0) < ( select comm
4                                 from emp
5                                where ename = 'WARD' )
```

Discussion

The COALESCE function will return the first non-NULL value from the list of values passed to it. When a NULL value is encountered, it is replaced by zero, which is then compared with WARD's commission. This can be seen by putting the COALESCE function in the SELECT list:

```
select ename,comm,coalesce(comm,0)
  from emp
 where coalesce(comm,0) < ( select comm
                                 from emp
                                where ename = 'WARD' )

ENAME            COMM COALESCE(COMM,0)
---------- ---------- ---------------
SMITH                               0
ALLEN             300             300
JONES                               0
BLAKE                               0
CLARK                               0
SCOTT                               0
KING                                0
TURNER              0               0
ADAMS                               0
JAMES                               0
FORD                                0
MILLER                              0
```

3.13 Summing Up

Joins are a crucial aspect of querying databases—it will be the norm that you need to join two or more tables together to find what you are looking for. Mastering the different combinations and categories of joins that are covered in this chapter will set you up for success.

CHAPTER 4

Inserting, Updating, and Deleting

The past few chapters have focused on basic query techniques, all centered around the task of getting data out of a database. This chapter turns the tables and focuses on the following three topic areas:

- Inserting new records into your database
- Updating existing records
- Deleting records that you no longer want

For ease in finding them when you need them, recipes in this chapter have been grouped by topic: all the insertion recipes come first, followed by the update recipes, and finally recipes for deleting data.

Inserting is usually a straightforward task. It begins with the simple problem of inserting a single row. Many times, however, it is more efficient to use a set-based approach to create new rows. To that end, you'll also find techniques for inserting many rows at a time.

Likewise, updating and deleting start out as simple tasks. You can update one record, and you can delete one record. But you can also update whole sets of records at once, and in very powerful ways. And there are many handy ways to delete records. For example, you can delete rows in one table depending on whether they exist in another table.

SQL even has a way, a relatively new addition to the standard, letting you insert, update, and delete all at once. That may not sound like too useful a thing now, but the MERGE statement represents a powerful way to synchronize a database table with an external source of data (such as a flat file feed from a remote system). Check out Recipe 4.11 in this chapter for details.

4.1 Inserting a New Record

Problem

You want to insert a new record into a table. For example, you want to insert a new record into the DEPT table. The value for DEPTNO should be 50, DNAME should be PROGRAMMING, and LOC should be BALTIMORE.

Solution

Use the INSERT statement with the VALUES clause to insert one row at a time:

```
insert into dept (deptno,dname,loc)
values (50,'PROGRAMMING','BALTIMORE')
```

For DB2, SQL Server, PostgreSQL, and MySQL you have the option of inserting one row at a time or multiple rows at a time by including multiple VALUES lists:

```
/* multi row insert */
insert into dept (deptno,dname,loc)
values (1,'A','B'),
       (2,'B','C')
```

Discussion

The INSERT statement allows you to create new rows in database tables. The syntax for inserting a single row is consistent across all database brands.

As a shortcut, you can omit the column list in an INSERT statement:

```
insert into dept
values (50,'PROGRAMMING','BALTIMORE')
```

However, if you do not list your target columns, you must insert into *all* of the columns in the table and be mindful of the order of the values in the VALUES list; you must supply values in the same order in which the database displays columns in response to a SELECT * query. Either way, you should be mindful of column constraints because if you don't insert into every column, you are will create a row where some values are null. This can cause an error if there are columns constrained not to accept nulls.

4.2 Inserting Default Values

Problem

A table can be defined to take default values for specific columns. You want to insert a row of default values without having to specify those values.

Consider the following table:

```
create table D (id integer default 0)
```

You want to insert zero without explicitly specifying zero in the values list of an INSERT statement. You want to explicitly insert the default, whatever that default is.

Solution

All brands support the use of the DEFAULT keyword as a way of explicitly specifying the default value for a column. Some brands provide additional ways to solve the problem.

The following example illustrates the use of the DEFAULT keyword:

```
insert into D values (default)
```

You may also explicitly specify the column name, which you'll need to do anytime you are not inserting into all columns of a table:

```
insert into D (id) values (default)
```

Oracle8*i* Database and prior versions do not support the DEFAULT keyword. Prior to Oracle9*i* Database, there was no way to explicitly insert a default column value.

MySQL allows you to specify an empty values list if all columns have a default value defined:

```
insert into D values ()
```

In this case, all columns will be set to their default values.

PostgreSQL and SQL Server support a DEFAULT VALUES clause:

```
insert into D default values
```

The DEFAULT VALUES clause causes all columns to take on their default values.

Discussion

The DEFAULT keyword in the values list will insert the value that was specified as the default for a particular column during table creation. The keyword is available for all DBMSs.

MySQL, PostgreSQL, and SQL Server users have another option available if all columns in the table are defined with a default value (as table D is in this case). You may use an empty VALUES list (MySQL) or specify the DEFAULT VALUES clause (PostgreSQL and SQL Server) to create a new row with all default values; otherwise, you need to specify DEFAULT for each column in the table.

For tables with a mix of default and nondefault columns, inserting default values for a column is as easy as excluding the column from the insert list; you do not need to use

the DEFAULT keyword. Say that table D had an additional column that was not defined with a default value:

```
create table D (id integer default 0, foo varchar(10))
```

You can insert a default for ID by listing only FOO in the insert list:

```
insert into D (name) values ('Bar')
```

This statement will result in a row in which ID is 0 and FOO is BAR. ID takes on its default value because no other value is specified.

4.3 Overriding a Default Value with NULL

Problem

You are inserting into a column having a default value, and you want to override that default value by setting the column to NULL. Consider the following table:

```
create table D (id integer default 0, foo VARCHAR(10))
```

You want to insert a row with a NULL value for ID.

Solution

You can explicitly specify NULL in your values list:

```
insert into d (id, foo) values (null, 'Brighten')
```

Discussion

Not everyone realizes that you can explicitly specify NULL in the values list of an INSERT statement. Typically, when you do not want to specify a value for a column, you leave that column out of your column and values lists:

```
insert into d (foo) values ('Brighten')
```

Here, no value for ID is specified. Many would expect the column to taken on the null value, but, alas, a default value was specified at table creation time, so the result of the preceding INSERT is that ID takes on the value zero (the default). By specifying NULL as the value for a column, you can set the column to NULL despite any default value (excepting where a constraint has been specifically applied to prevent NULLs).

4.4 Copying Rows from One Table into Another

Problem

You want to copy rows from one table to another by using a query. The query may be complex or simple, but ultimately you want the result to be inserted into another

table. For example, you want to copy rows from the DEPT table to the DEPT_EAST table. The DEPT_EAST table has already been created with the same structure (same columns and data types) as DEPT and is currently empty.

Solution

Use the INSERT statement followed by a query to produce the rows you want:

```
1 insert into dept_east (deptno,dname,loc)
2 select deptno,dname,loc
3   from dept
4  where loc in ( 'NEW YORK','BOSTON' )
```

Discussion

Simply follow the INSERT statement with a query that returns the desired rows. If you want to copy all rows from the source table, exclude the WHERE clause from the query. Like a regular insert, you do not have to explicitly specify which columns you are inserting into. But if you do not specify your target columns, you must insert data into *all* of the table's columns, and you must be mindful of the order of the values in the SELECT list, as described earlier in Recipe 4.1.

4.5 Copying a Table Definition

Problem

You want to create a new table having the same set of columns as an existing table. For example, you want to create a copy of the DEPT table and call it DEPT_2. You do not want to copy the rows, only the column structure of the table.

Solution

DB2

Use the LIKE clause with the CREATE TABLE command:

```
create table dept_2 like dept
```

Oracle, MySQL, and PostgreSQL

Use the CREATE TABLE command with a subquery that returns no rows:

```
1 create table dept_2
2 as
3 select *
4   from dept
5  where 1 = 0
```

SQL Server

Use the INTO clause with a subquery that returns no rows:

```
1 select *
2   into dept_2
3   from dept
4  where 1 = 0
```

Discussion

DB2

DB2's CREATE TABLE…LIKE command allows you to easily use one table as the pattern for creating another. Simply specify your pattern table's name following the LIKE keyword.

Oracle, MySQL, and PostgreSQL

When using Create Table As Select (CTAS), all rows from your query will be used to populate the new table you are creating unless you specify a false condition in the WHERE clause. In the solution provided, the expression "1 = 0" in the WHERE clause of the query causes no rows to be returned. Thus, the result of the CTAS statement is an empty table based on the columns in the SELECT clause of the query.

SQL Server

When using INTO to copy a table, all rows from your query will be used to populate the new table you are creating unless you specify a false condition in the WHERE clause of your query. In the solution provided, the expression "1 = 0" in the predicate of the query causes no rows to be returned. The result is an empty table based on the columns in the SELECT clause of the query.

4.6 Inserting into Multiple Tables at Once

Problem

You want to take rows returned by a query and insert those rows into multiple target tables. For example, you want to insert rows from DEPT into tables DEPT_EAST, DEPT_WEST, and DEPT_MID. All three tables have the same structure (same columns and data types) as DEPT and are currently empty.

Solution

The solution is to insert the result of a query into the target tables. The difference from Recipe 4.4 is that for this problem you have multiple target tables.

Oracle

Use either the INSERT ALL or INSERT FIRST statement. Both share the same syntax except for the choice between the ALL and FIRST keywords. The following statement uses INSERT ALL to cause all possible target tables to be considered:

```
1   insert all
2     when loc in ('NEW YORK','BOSTON') then
3     into dept_east (deptno,dname,loc) values (deptno,dname,loc)
4     when loc = 'CHICAGO' then
5       into dept_mid (deptno,dname,loc) values (deptno,dname,loc)
6     else
7       into dept_west (deptno,dname,loc) values (deptno,dname,loc)
8     select deptno,dname,loc
9       from dept
```

DB2

Insert into an inline view that performs a UNION ALL on the tables to be inserted. You must also be sure to place constraints on the tables that will ensure each row goes into the correct table:

```
create table dept_east
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc in ('NEW YORK','BOSTON')))

create table dept_mid
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc = 'CHICAGO'))

create table dept_west
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc = 'DALLAS'))

1  insert into (
2    select * from dept_west union all
3    select * from dept_east union all
4    select * from dept_mid
5  ) select * from dept
```

MySQL, PostgreSQL, and SQL Server

As of the time of this writing, these vendors do not support multitable inserts.

Discussion

Oracle

Oracle's multitable insert uses WHEN-THEN-ELSE clauses to evaluate the rows from the nested SELECT and insert them accordingly. In this recipe's example, INSERT ALL and INSERT FIRST would produce the same result, but there is a difference between the two. INSERT FIRST will break out of the WHEN-THEN-ELSE evaluation as soon as it encounters a condition evaluating to true; INSERT ALL will evaluate all conditions even if prior tests evaluate to true. Thus, you can use INSERT ALL to insert the same row into more than one table.

DB2

My DB2 solution is a bit of a hack. It requires that the tables to be inserted into have constraints defined to ensure that each row evaluated from the subquery will go into the correct table. The technique is to insert into a view that is defined as the UNION ALL of the tables. If the check constraints are not unique among the tables in the INSERT (i.e., multiple tables have the same check constraint), the INSERT statement will not know where to put the rows, and it will fail.

MySQL, PostgreSQL, and SQL Server

As of the time of this writing, only Oracle and DB2 provide mechanisms to insert rows returned by a query into one or more of several tables within the same statement.

4.7 Blocking Inserts to Certain Columns

Problem

You want to prevent users, or an errant software application, from inserting values into certain table columns. For example, you want to allow a program to insert into EMP, but only into the EMPNO, ENAME, and JOB columns.

Solution

Create a view on the table exposing only those columns you want to expose. Then force all inserts to go through that view.

For example, to create a view exposing the three columns in EMP:

```
create view new_emps as
select empno, ename, job
  from emp
```

Grant access to this view to those users and programs allowed to populate only the three fields in the view. Do not grant those users insert access to the EMP table. Users may then create new EMP records by inserting into the NEW_EMPS view, but they will not be able to provide values for columns other than the three that are specified in the view definition.

Discussion

When you insert into a simple view such as in the solution, your database server will translate that insert into the underlying table. For example, the following insert:

```
insert into new_emps
   (empno ename, job)
   values (1, 'Jonathan', 'Editor')
```

will be translated behind the scenes into:

```
insert into emp
   (empno ename, job)
   values (1, 'Jonathan', 'Editor')
```

It is also possible, but perhaps less useful, to insert into an inline view (currently only supported by Oracle):

```
insert into
  (select empno, ename, job
     from emp)
values (1, 'Jonathan', 'Editor')
```

View insertion is a complex topic. The rules become complicated very quickly for all but the simplest of views. If you plan to make use of the ability to insert into views, it is imperative that you consult and fully understand your vendor documentation on the matter.

4.8 Modifying Records in a Table

Problem

You want to modify values for some or all rows in a table. For example, you might want to increase the salaries of everyone in department 20 by 10%. The following result set shows the DEPTNO, ENAME, and SAL for employees in that department:

```
select deptno,ename,sal
  from emp
 where deptno = 20
 order by 1,3

DEPTNO ENAME             SAL
------ ---------- ----------
    20 SMITH             800
```

```
    20 ADAMS            1100
    20 JONES            2975
    20 SCOTT            3000
    20 FORD             3000
```

You want to bump all the SAL values by 10%.

Solution

Use the UPDATE statement to modify existing rows in a database table. For example:

```
update emp
   set sal = sal*1.10
 where deptno = 20
```

Discussion

Use the UPDATE statement along with a WHERE clause to specify which rows to update; if you exclude a WHERE clause, then all rows are updated. The expression SAL*1.10 in this solution returns the salary increase by 10%.

When preparing for a mass update, you may want to preview the results. You can do that by issuing a SELECT statement that includes the expressions you plan to put into your SET clauses. The following SELECT shows the result of a 10% salary increase:

```
select deptno,
       ename,
       sal      as orig_sal,
       sal*.10  as amt_to_add,
       sal*1.10 as new_sal
  from emp
 where deptno=20
 order by 1,5

DEPTNO ENAME  ORIG_SAL AMT_TO_ADD  NEW_SAL
------ ------ -------- ----------  -------
    20 SMITH       800         80      880
    20 ADAMS      1100        110     1210
    20 JONES      2975        298     3273
    20 SCOTT      3000        300     3300
    20 FORD       3000        300     3300
```

The salary increase is broken down into two columns: one to show the increase over the old salary, and the other to show the new salary.

4.9 Updating When Corresponding Rows Exist

Problem

You want to update rows in one table when corresponding rows exist in another. For example, if an employee appears in table EMP_BONUS, you want to increase that employee's salary (in table EMP) by 20%. The following result set represents the data currently in table EMP_BONUS:

```
select empno, ename
  from emp_bonus

     EMPNO ENAME
---------- ---------
      7369 SMITH
      7900 JAMES
      7934 MILLER
```

Solution

Use a subquery in your UPDATE statement's WHERE clause to find employees in table EMP that are also in table EMP_BONUS. Your UPDATE will then act only on those rows, enabling you to increase their salary by 20%:

```
1 update emp
2    set sal=sal*1.20
3  where empno in ( select empno from emp_bonus )
```

Discussion

The results from the subquery represent the rows that will be updated in table EMP. The IN predicate tests values of EMPNO from the EMP table to see whether they are in the list of EMPNO values returned by the subquery. When they are, the corresponding SAL values are updated.

Alternatively, you can use EXISTS instead of IN:

```
update emp
   set sal = sal*1.20
 where exists ( select null
                  from emp_bonus
                 where emp.empno=emp_bonus.empno )
```

You may be surprised to see NULL in the SELECT list of the EXISTS subquery. Fear not, that NULL does not have an adverse effect on the update. Arguably it increases readability as it reinforces the fact that, unlike the solution using a subquery with an IN operator, what will drive the update (i.e., which rows will be updated) will be controlled by the WHERE clause of the subquery, not the values returned as a result of the subquery's SELECT list.

4.10 Updating with Values from Another Table

Problem

You want to update rows in one table using values from another. For example, you have a table called NEW_SAL, which holds the new salaries for certain employees. The contents of table NEW_SAL are as follows:

```
select *
  from new_sal

DEPTNO        SAL
------ ----------
    10       4000
```

Column DEPTNO is the primary key of table NEW_SAL. You want to update the salaries and commission of certain employees in table EMP using values table NEW_SAL if there is a match between EMP.DEPTNO and NEW_SAL.DEPTNO, update EMP.SAL to NEW_SAL.SAL, and update EMP.COMM to 50% of NEW_SAL.SAL. The rows in EMP are as follows:

```
select deptno,ename,sal,comm
  from emp
 order by 1

DEPTNO ENAME             SAL       COMM
------ ---------- ---------- ----------
    10 CLARK           2450
    10 KING            5000
    10 MILLER          1300
    20 SMITH            800
    20 ADAMS           1100
    20 FORD            3000
    20 SCOTT           3000
    20 JONES           2975
    30 ALLEN           1600        300
    30 BLAKE           2850
    30 MARTIN          1250       1400
    30 JAMES            950
    30 TURNER          1500          0
    30 WARD            1250        500
```

Solution

Use a join between NEW_SAL and EMP to find and return the new COMM values to the UPDATE statement. It is quite common for updates such as this one to be performed via correlated subquery or alternatively using a CTE. Another technique involves creating a view (traditional or inline, depending on what your database supports) and then updating that view.

DB2

Use a correlated subquery to set new SAL and COMM values in EMP. Also use a correlated subquery to identify which rows from EMP should be updated:

```
update emp e set (e.sal,e.comm) = (select ns.sal, ns.sal/2
                                    from new_sal ns
                                   where ns.deptno=e.deptno)
 where exists ( select *
                  from new_sal ns
                 where ns.deptno = e.deptno )
```

MySQL

Include both EMP and NEW_SAL in the UPDATE clause of the UPDATE statement and join in the WHERE clause:

```
update emp e, new_sal ns
set e.sal=ns.sal,
e.comm=ns.sal/2
where e.deptno=ns.deptno
```

Oracle

The method for the DB2 solution will work for Oracle, but as an alternative, you can update an inline view:

```
update (
 select e.sal as emp_sal, e.comm as emp_comm,
        ns.sal as ns_sal, ns.sal/2 as ns_comm
   from emp e, new_sal ns
  where e.deptno = ns.deptno
) set emp_sal = ns_sal, emp_comm = ns_comm
```

PostgreSQL

The method used for the DB2 solution will work for PostgreSQL, but you could also (quite conveniently) join directly in the UPDATE statement:

```
update emp
   set sal = ns.sal,
       comm = ns.sal/2
  from new_sal ns
 where ns.deptno = emp.deptno
```

SQL Server

The method used for the DB2 solution will work for SQL Server, but as an alternative you can (similarly to the PostgreSQL solution) join directly in the UPDATE statement:

```
1 update e
2    set e.sal  = ns.sal,
3        e.comm = ns.sal/2
4   from emp e,
5        new_sal ns
6  where ns.deptno = e.deptno
```

Discussion

Before discussing the different solutions, it's worth mentioning something important regarding updates that use queries to supply new values. A WHERE clause in the subquery of a correlated update is not the same as the WHERE clause of the table being updated. If you look at the UPDATE statement in the "Problem" section, the join on DEPTNO between EMP and NEW_SAL is done and returns rows to the SET clause of the UPDATE statement. For employees in DEPTNO 10, valid values are returned because there is a matching DEPTNO in table NEW_SAL. But what about employees in the other departments? NEW_SAL does not have any other departments, so the SAL and COMM for employees in DEPTNOs 20 and 30 are set to NULL. Unless you are doing so via LIMIT or TOP or whatever mechanism your vendor supplies for limiting the number of rows returned in a result set, the only way to restrict rows from a table in SQL is to use a WHERE clause. To correctly perform this UPDATE, use a WHERE clause on the table being updated along with a WHERE clause in the correlated subquery.

DB2

To ensure you do not update every row in table EMP, remember to include a correlated subquery in the WHERE clause of the UPDATE. Performing the join (the correlated subquery) in the SET clause is not enough. By using a WHERE clause in the UPDATE, you ensure that only rows in EMP that match on DEPTNO to table NEW_SAL are updated. This holds true for all RDBMSs.

Oracle

In the Oracle solution using the update join view, you are using equi-joins to determine which rows will be updated. You can confirm which rows are being updated by executing the query independently. To be able to successfully use this type of UPDATE, you must first understand the concept of key-preservation. The DEPTNO column of the table NEW_SAL is the primary key of that table; thus, its values are unique within the table. When joining between EMP and NEW_SAL, however, NEW_SAL.DEPTNO is not unique in the result set, as shown here:

```
select e.empno, e.deptno e_dept, ns.sal, ns.deptno ns_deptno
  from emp e, new_sal ns
 where e.deptno = ns.deptno
```

```
EMPNO     E_DEPT        SAL  NS_DEPTNO
----- ---------- ---------- ----------
 7782         10       4000         10
 7839         10       4000         10
 7934         10       4000         10
```

To enable Oracle to update this join, one of the tables must be key-preserved, meaning that if its values are not unique in the result set, it should at least be unique in the table it comes from. In this case, NEW_SAL has a primary key on DEPTNO, which makes it unique in the table. Because it is unique in its table, it may appear multiple times in the result set and will still be considered key-preserved, thus allowing the update to complete successfully.

PostgreSQL, SQL Server, and MySQL

The syntax is a bit different between these platforms, but the technique is the same. Being able to join directly in the UPDATE statement is extremely convenient. Since you specify which table to update (the table listed after the UPDATE keyword), there's no confusion as to which table's rows are modified. Additionally, because you are using joins in the update (since there is an explicit WHERE clause), you can avoid some of the pitfalls when coding correlated subquery updates; in particular, if you missed a join here, it would be obvious you'd have a problem.

4.11 Merging Records

Problem

You want to conditionally insert, update, or delete records in a table depending on whether corresponding records exist. (If a record exists, then update; if not, then insert; if after updating a row fails to meet a certain condition, delete it.) For example, you want to modify table EMP_COMMISSION such that:

- If any employee in EMP_COMMISSION also exists in table EMP, then update their commission (COMM) to 1000.
- For all employees who will potentially have their COMM updated to 1000, if their SAL is less than 2000, delete them (they should not be exist in EMP_[.keep-together] COMMISSION).
- Otherwise, insert the EMPNO, ENAME, and DEPTNO values from table EMP into table EMP_COMMISSION.

Essentially, you want to execute either an UPDATE or an INSERT depending on whether a given row from EMP has a match in EMP_COMMISSION. Then you want to execute a DELETE if the result of an UPDATE causes a commission that's too high.

The following rows are currently in tables EMP and EMP_COMMISSION, respectively:

```
select deptno,empno,ename,comm
  from emp
 order by 1

DEPTNO      EMPNO ENAME        COMM
------ ---------- ------ ----------
    10       7782  CLARK
    10       7839  KING
    10       7934  MILLER
    20       7369  SMITH
    20       7876  ADAMS
    20       7902  FORD
    20       7788  SCOTT
    20       7566  JONES
    30       7499  ALLEN        300
    30       7698  BLAKE
    30       7654  MARTIN      1400
    30       7900  JAMES
    30       7844  TURNER         0
    30       7521  WARD         500


select deptno,empno,ename,comm
  from emp_commission
 order by 1

    DEPTNO      EMPNO ENAME            COMM
---------- ---------- ---------- ----------
        10       7782 CLARK
        10       7839 KING
        10       7934 MILLER
```

Solution

The statement designed to solve this problem is the MERGE statement, and it can perform either an UPDATE or an INSERT, as needed. For example:

```
1  merge into emp_commission ec
2  using (select * from emp) emp
3     on (ec.empno=emp.empno)
4   when matched then
5        update set ec.comm = 1000
6        delete where (sal < 2000)
7   when not matched then
8        insert (ec.empno,ec.ename,ec.deptno,ec.comm)
9        values (emp.empno,emp.ename,emp.deptno,emp.comm)
```

Currently, MySQL does not have a MERGE statement; otherwise, this query should work on any RDBMS in this book, and in a wide number of others.

Discussion

The join on line 3 of the solution determines what rows already exist and will be updated. The join is between EMP_COMMISSION (aliased as EC) and the subquery (aliased as EMP). When the join succeeds, the two rows are considered "matched," and the UPDATE specified in the WHEN MATCHED clause is executed. Otherwise, no match is found, and the INSERT in WHEN NOT MATCHED is executed. Thus, rows from table EMP that do not have corresponding rows based on EMPNO in table EMP_COMMISSION will be inserted into EMP_COMMISSION. Of all the employees in table EMP, only those in DEPTNO 10 should have their COMM updated in EMP_COMMISSION, while the rest of the employees are inserted. Additionally, since MILLER is in DEPTNO 10, he is a candidate to have his COMM updated, but because his SAL is less than 2,000, it is deleted from EMP_COMMISSION.

4.12 Deleting All Records from a Table

Problem

You want to delete all the records from a table.

Solution

Use the DELETE command to delete records from a table. For example, to delete all records from EMP, use the following:

```
delete from emp
```

Discussion

When using the DELETE command without a WHERE clause, you will delete all rows from the table specified. Sometimes TRUNCATE, which applies to tables and therefore doesn't use the WHERE clause, is preferred as it is faster. At least in Oracle, however, TRUNCATE cannot be undone. You should carefully check vendor documentation for a detailed view of the performance and rollback differences between TRUNCATE and DELETE in your specific RDBMS.

4.13 Deleting Specific Records

Problem

You want to delete records meeting a specific criterion from a table.

Solution

Use the DELETE command with a WHERE clause specifying which rows to delete. For example, to delete all employees in department 10, use the following:

```
delete from emp where deptno = 10
```

Discussion

By using a WHERE clause with the DELETE command, you can delete a subset of rows in a table rather than all the rows. Don't forget to check that you're deleting the right data by previewing the effect of your WHERE clause using SELECT—you can delete the wrong data even in a simple situation. For example, in the previous case, a typo could lead to the employees in department 20 being deleted instead of department 10!

4.14 Deleting a Single Record

Problem

You want to delete a single record from a table.

Solution

This is a special case of Recipe 4.13. The key is to ensure that your selection criterion is narrow enough to specify only the one record that you want to delete. Often you will want to delete based on the primary key. For example, to delete employee CLARK (EMPNO 7782):

```
delete from emp where empno = 7782
```

Discussion

Deleting is always about identifying the rows to be deleted, and the impact of a DELETE always comes down to its WHERE clause. Omit the WHERE clause and the scope of a DELETE is the entire table. By writing conditions in the WHERE clause, you can narrow the scope to a group of records or to a single record. When deleting a single record, you should typically be identifying that record based on its primary key or on one of its unique keys.

If your deletion criterion is based on a primary or unique key, then you can be sure of deleting only one record. (This is because your RDBMS will not allow two rows to contain the same primary or unique key values.) Otherwise, you may want to check first, to be sure you aren't about to inadvertently delete more records than you intend.

4.15 Deleting Referential Integrity Violations

Problem

You want to delete records from a table when those records refer to nonexistent records in some other table. For example, some employees are assigned to departments that do not exist. You want to delete those employees.

Solution

Use the NOT EXISTS predicate with a subquery to test the validity of department numbers:

```
delete from emp
 where not exists (
   select * from dept
    where dept.deptno = emp.deptno
)
```

Alternatively, you can write the query using a NOT IN predicate:

```
delete from emp
where deptno not in (select deptno from dept)
```

Discussion

Deleting is really all about selecting: the real work lies in writing WHERE clause conditions to correctly describe those records that you want to delete.

The NOT EXISTS solution uses a correlated subquery to test for the existence of a record in DEPT having a DEPTNO matching that in a given EMP record. If such a record exists, then the EMP record is retained. Otherwise, it is deleted. Each EMP record is checked in this manner.

The IN solution uses a subquery to retrieve a list of valid department numbers. DEPTNOs from each EMP record are then checked against that list. When an EMP record is found with a DEPTNO not in the list, the EMP record is deleted.

4.16 Deleting Duplicate Records

Problem

You want to delete duplicate records from a table. Consider the following table:

```
create table dupes (id integer, name varchar(10))

insert into dupes values (1, 'NAPOLEON')
insert into dupes values (2, 'DYNAMITE')
```

```
insert into dupes values (3, 'DYNAMITE')
insert into dupes values (4, 'SHE SELLS')
insert into dupes values (5, 'SEA SHELLS')
insert into dupes values (6, 'SEA SHELLS')
insert into dupes values (7, 'SEA SHELLS')

select * from dupes order by 1

        ID NAME
---------- ----------
         1 NAPOLEON
         2 DYNAMITE
         3 DYNAMITE
         4 SHE SELLS
         5 SEA SHELLS
         6 SEA SHELLS
         7 SEA SHELLS
```

For each group of duplicate names, such as SEA SHELLS, you want to arbitrarily retain one ID and delete the rest. In the case of SEA SHELLS, you don't care whether you delete lines 5 and 6, or lines 5 and 7, or lines 6 and 7, but in the end you want just one record for SEA SHELLS.

Solution

Use a subquery with an aggregate function such as MIN to arbitrarily choose the ID to retain (in this case only the NAME with the smallest value for ID is not deleted):

```
1  delete from dupes
2   where id not in ( select min(id)
3                       from dupes
4                      group by name )
```

For MySQL users you will need slightly different syntax because you cannot reference the same table twice in a delete (as of the time of this writing):

```
1   delete from dupes
2    where id not in
3          (select min(id)
4    from (select id,name from dupes) tmp
5           group by name)
```

Discussion

The first thing to do when deleting duplicates is to define exactly what it means for two rows to be considered "duplicates" of each other. For my example in this recipe, the definition of "duplicate" is that two records contain the same value in their NAME column. Having that definition in place, you can look to some other column to discriminate among each set of duplicates, to identify those records to retain. It's best if

this discriminating column (or columns) is a primary key. We used the ID column, which is a good choice because no two records have the same ID.

The key to the solution is that you group by the values that are duplicated (by NAME in this case), and then use an aggregate function to pick off just one key value to retain. The subquery in the "Solution" example will return the smallest ID for each NAME, which represents the row you will not delete:

```
select min(id)
  from dupes
 group by name

    MIN(ID)
-----------
          2
          1
          5
          4
```

The DELETE then deletes any ID in the table that is not returned by the subquery (in this case IDs 3, 6, and 7). If you are having trouble seeing how this works, run the subquery first and include the NAME in the SELECT list:

```
select name, min(id)
  from dupes
 group by name

NAME          MIN(ID)
---------- ----------
DYNAMITE           2
NAPOLEON           1
SEA SHELLS         5
SHE SELLS          4
```

The rows returned by the subquery represent those to be retained. The NOT IN predicate in the DELETE statement causes all other rows to be deleted.

4.17 Deleting Records Referenced from Another Table

Problem

You want to delete records from one table when those records are referenced from some other table. Consider the following table, named DEPT_ACCIDENTS, which contains one row for each accident that occurs in a manufacturing business. Each row records the department in which an accident occurred and also the type of accident.

```
create table dept_accidents
( deptno        integer,
  accident_name  varchar(20) )
```

```
insert into dept_accidents values (10,'BROKEN FOOT')
insert into dept_accidents values (10,'FLESH WOUND')
insert into dept_accidents values (20,'FIRE')
insert into dept_accidents values (20,'FIRE')
insert into dept_accidents values (20,'FLOOD')
insert into dept_accidents values (30,'BRUISED GLUTE')

select * from dept_accidents

    DEPTNO ACCIDENT_NAME
---------- --------------------
        10 BROKEN FOOT
        10 FLESH WOUND
        20 FIRE
        20 FIRE
        20 FLOOD
        30 BRUISED GLUTE
```

You want to delete from EMP the records for those employees working at a department that has three or more accidents.

Solution

Use a subquery and the aggregate function COUNT to find the departments with three or more accidents. Then delete all employees working in those departments:

```
1 delete from emp
2  where deptno in ( select deptno
3                      from dept_accidents
4                     group by deptno
5                    having count(*) >= 3 )
```

Discussion

The subquery will identify which departments have three or more accidents:

```
select deptno
  from dept_accidents
 group by deptno
having count(*) >= 3

    DEPTNO
----------
        20
```

The DELETE will then delete any employees in the departments returned by the subquery (in this case, only in department 20).

4.18 Summing Up

Inserting and updating data may seem to take up less of your time than querying data, and in the rest of the book we will concentrate on queries. However, being able to maintain the data in a database is clearly fundamental to its purpose, and these recipes are a crucial part of the skill set needed to maintain a database. Some of these commands, especially commands that remove or delete data, can have lasting consequences. Always preview any data you intend to delete to make sure you are really deleting what you mean to, and become familiar with what can and can't be undone in your specific RDBMS.

CHAPTER 5

Metadata Queries

This chapter presents recipes that allow you to find information about a given schema. For example, you may want to know what tables you've created or which foreign keys are not indexed. All of the RDBMSs in this book provide tables and views for obtaining such data. The recipes in this chapter will get you started on gleaning information from those tables and views.

Although at a high level the strategy of storing metadata in tables and views within the RDBMS is common, the ultimate implementation is not standardized to the same degree as most of the SQL language features covered in this book. Therefore, compared to other chapters, in this chapter having a different solution for each RDBMS is far more common.

The following is selection of the most common schema queries written for each of the RDMSs covered in the book. There is far more information available than the recipes in this chapter can show. Consult your RDBMS's documentation for the complete list of catalog or data dictionary tables/views when you need to go beyond what's presented here.

For the purposes of demonstration, all of the recipes in this chapter assume there is a schema named SMEAGOL.

5.1 Listing Tables in a Schema

Problem

You want to see a list of all the tables you've created in a given schema.

Solution

The solutions that follow all assume you are working with the SMEAGOL schema. The basic approach to a solution is the same for all RDBMSs: you query a system table (or view) containing a row for each table in the database.

DB2

Query SYSCAT.TABLES:

```
1 select tabname
2   from syscat.tables
3  where tabschema = 'SMEAGOL'
```

Oracle

Query SYS.ALL_TABLES:

```
select table_name
  from all_tables
 where owner = 'SMEAGOL'
```

PostgreSQL, MySQL, and SQL Server

Query INFORMATION_SCHEMA.TABLES:

```
1 select table_name
2   from information_schema.tables
3  where table_schema = 'SMEAGOL'
```

Discussion

In a delightfully circular manner, databases expose information about themselves through the very mechanisms that you create for your own applications: tables and views. Oracle, for example, maintains an extensive catalog of system views, such as ALL_TABLES, that you can query for information about tables, indexes, grants, and any other database object.

Oracle's catalog views are just that, views. They are based on an underlying set of tables that contain the information in a user-unfriendly form. The views put a usable face on Oracle's catalog data.

Oracle's system views and DB2's system tables are each vendor-specific. PostgreSQL, MySQL, and SQL Server, on the other hand, support something called the *information* schema, which is a set of views defined by the ISO SQL standard. That's why the same query can work for all three of those databases.

5.2 Listing a Table's Columns

Problem

You want to list the columns in a table, along with their data types, and their position in the table they are in.

Solution

The following solutions assume that you want to list columns, their data types, and their numeric position in the table named EMP in the schema SMEAGOL.

DB2

Query SYSCAT.COLUMNS:

```
1 select colname, typename, colno
2   from syscat.columns
3  where tabname   = 'EMP'
4    and tabschema = 'SMEAGOL'
```

Oracle

Query ALL_TAB_COLUMNS:

```
1 select column_name, data_type, column_id
2   from all_tab_columns
3  where owner      = 'SMEAGOL'
4    and table_name = 'EMP'
```

PostgreSQL, MySQL, and SQL Server

Query INFORMATION_SCHEMA.COLUMNS:

```
1 select column_name, data_type, ordinal_position
2   from information_schema.columns
3  where table_schema = 'SMEAGOL'
4    and table_name   = 'EMP'
```

Discussion

Each vendor provides ways for you to get detailed information about your column data. In the previous examples, only the column name, data type, and position are returned. Additional useful items of information include length, nullability, and default values.

5.3 Listing Indexed Columns for a Table

Problem

You want list indexes, their columns, and the column position (if available) in the index for a given table.

Solution

The vendor-specific solutions that follow all assume that you are listing indexes for table EMP in the SMEAGOL schema.

DB2

Query SYSCAT.INDEXES:

```
select a.tabname, b.indname, b.colname, b.colseq
  from syscat.indexes a,
       syscat.indexcoluse b
 where a.tabname   = 'EMP'
   and a.tabschema = 'SMEAGOL'
   and a.indschema = b.indschema
   and a.indname   = b.indname
```

Oracle

Query SYS.ALL_IND_COLUMNS:

```
select table_name, index_name, column_name, column_position
  from sys.all_ind_columns
 where table_name  = 'EMP'
   and table_owner = 'SMEAGOL'
```

PostgreSQL

Query PG_CATALOG.PG_INDEXES and INFORMATION_SCHEMA.COLUMNS:

```
select a.tablename,a.indexname,b.column_name
  from pg_catalog.pg_indexes a,
       information_schema.columns b
 where a.schemaname = 'SMEAGOL'
   and a.tablename  = b.table_name
```

MySQL

Use the SHOW INDEX command:

```
show index from emp
```

SQL Server

Query SYS.TABLES, SYS.INDEXES, SYS.INDEX_COLUMNS, and SYS.COLUMNS:

```
select a.name table_name,
       b.name index_name,
       d.name column_name,
       c.index_column_id
  from sys.tables a,
       sys.indexes b,
       sys.index_columns c,
       sys.columns d
where a.object_id = b.object_id
  and b.object_id = c.object_id
  and b.index_id  = c.index_id
  and c.object_id = d.object_id
  and c.column_id = d.column_id
  and a.name      = 'EMP'
```

Discussion

When it comes to queries, it's important to know what columns are/aren't indexed. Indexes can provide good performance for queries against columns that are frequently used in filters and that are fairly selective. Indexes are also useful when joining between tables. By knowing what columns are indexed, you are already one step ahead of performance problems if they should occur. Additionally, you might want to find information about the indexes themselves: how many levels deep they are, how many distinct keys there are, how many leaf blocks there are, and so forth. Such information is also available from the views/tables queried in this recipe's solutions.

5.4 Listing Constraints on a Table

Problem

You want to list the constraints defined for a table in some schema and the columns they are defined on. For example, you want to find the constraints and the columns they are on for table EMP.

Solution

DB2

Query SYSCAT.TABCONST and SYSCAT.COLUMNS:

```
select a.tabname, a.constname, b.colname, a.type
  from syscat.tabconst a,
       syscat.columns b
where a.tabname   = 'EMP'
```

```
5    and a.tabschema = 'SMEAGOL'
6    and a.tabname   = b.tabname
7    and a.tabschema = b.tabschema
```

Oracle

Query SYS.ALL_CONSTRAINTS and SYS.ALL_CONS_COLUMNS:

```
 1  select a.table_name,
 2         a.constraint_name,
 3         b.column_name,
 4         a.constraint_type
 5    from all_constraints a,
 6         all_cons_columns b
 7  where a.table_name      = 'EMP'
 8    and a.owner           = 'SMEAGOL'
 9    and a.table_name      = b.table_name
10    and a.owner           = b.owner
11    and a.constraint_name = b.constraint_name
```

PostgreSQL, MySQL, and SQL Server

Query INFORMATION_SCHEMA.TABLE_CONSTRAINTS and INFORMATION_SCHEMA.KEY_COLUMN_USAGE:

```
 1  select a.table_name,
 2         a.constraint_name,
 3         b.column_name,
 4         a.constraint_type
 5    from information_schema.table_constraints a,
 6         information_schema.key_column_usage b
 7  where a.table_name      = 'EMP'
 8    and a.table_schema    = 'SMEAGOL'
 9    and a.table_name      = b.table_name
10    and a.table_schema    = b.table_schema
11    and a.constraint_name = b.constraint_name
```

Discussion

Constraints are such a critical part of relational databases that it should go without saying why you need to know what constraints are on your tables. Listing the constraints on tables is useful for a variety of reasons: you may want to find tables missing a primary key, you may want to find which columns should be foreign keys but are not (i.e., child tables have data different from the parent tables and you want to know how that happened), or you may want to know about check constraints (Are columns nullable? Do they have to satisfy a specific condition? etc.).

5.5 Listing Foreign Keys Without Corresponding Indexes

Problem

You want to list tables that have foreign key columns that are not indexed. For example, you want to determine whether the foreign keys on table EMP are indexed.

Solution

DB2

Query SYSCAT.TABCONST, SYSCAT.KEYCOLUSE, SYSCAT.INDEXES, and SYSCAT.INDEXCOLUSE:

```
select fkeys.tabname,
       fkeys.constname,
       fkeys.colname,
       ind_cols.indname
  from (
select a.tabschema, a.tabname, a.constname, b.colname
  from syscat.tabconst a,
       syscat.keycoluse b
where a.tabname    = 'EMP'
  and a.tabschema  = 'SMEAGOL'
  and a.type       = 'F'
  and a.tabname    = b.tabname
  and a.tabschema  = b.tabschema
      ) fkeys
      left join
      (
select a.tabschema,
       a.tabname,
       a.indname,
       b.colname
  from syscat.indexes a,
       syscat.indexcoluse b
where a.indschema  = b.indschema
  and a.indname    = b.indname
      ) ind_cols
   on (fkeys.tabschema = ind_cols.tabschema
         and fkeys.tabname   = ind_cols.tabname
         and fkeys.colname   = ind_cols.colname )
where ind_cols.indname is null
```

Oracle

Query SYS.ALL_CONS_COLUMNS, SYS.ALL_CONSTRAINTS, and SYS.ALL_IND_COLUMNS:

```
select a.table_name,
       a.constraint_name,
       a.column_name,
       c.index_name
  from all_cons_columns a,
       all_constraints b,
       all_ind_columns c
where a.table_name      = 'EMP'
  and a.owner           = 'SMEAGOL'
  and b.constraint_type = 'R'
  and a.owner           = b.owner
  and a.table_name      = b.table_name
  and a.constraint_name = b.constraint_name
  and a.owner           = c.table_owner (+)
  and a.table_name      = c.table_name (+)
  and a.column_name     = c.column_name (+)
  and c.index_name      is null
```

PostgreSQL

Query INFORMATION_SCHEMA.KEY_COLUMN_USAGE, INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS, INFORMATION_SCHEMA.COLUMNS, and PG_CATALOG.PG_INDEXES:

```
select fkeys.table_name,
       fkeys.constraint_name,
       fkeys.column_name,
       ind_cols.indexname
  from (
select a.constraint_schema,
       a.table_name,
       a.constraint_name,
       a.column_name
  from information_schema.key_column_usage a,
       information_schema.referential_constraints b
 where a.constraint_name   = b.constraint_name
   and a.constraint_schema = b.constraint_schema
   and a.constraint_schema = 'SMEAGOL'
   and a.table_name        = 'EMP'
       ) fkeys
       left join
       (
select a.schemaname, a.tablename, a.indexname, b.column_name
  from pg_catalog.pg_indexes a,
       information_schema.columns b
 where a.tablename  = b.table_name
   and a.schemaname = b.table_schema
       ) ind_cols
    on (  fkeys.constraint_schema = ind_cols.schemaname
          and fkeys.table_name     = ind_cols.tablename
          and fkeys.column_name    = ind_cols.column_name )
 where ind_cols.indexname is null
```

MySQL

You can use the SHOW INDEX command to retrieve index information such as index name, columns in the index, and ordinal position of the columns in the index. Additionally, you can query INFORMATION_SCHEMA.KEY_COLUMN_USAGE to list the foreign keys for a given table. In MySQL 5, foreign keys are said to be indexed automatically, but can in fact be dropped. To determine whether a foreign key column's index has been dropped, you can execute SHOW INDEX for a particular table and compare the output with that of INFORMATION_SCHEMA.KEY_COLUMN_USAGE.COLUMN_NAME for the same table. If the COLUMN_NAME is listed in KEY_COLUMN_USAGE but is not returned by SHOW INDEX, you know that column is not indexed.

SQL Server

Query SYS.TABLES, SYS.FOREIGN_KEYS, SYS.COLUMNS, SYS.INDEXES, and SYS.INDEX_COLUMNS:

```
select fkeys.table_name,
       fkeys.constraint_name,
       fkeys.column_name,
       ind_cols.index_name
  from (
select a.object_id,
       d.column_id,
       a.name table_name,
       b.name constraint_name,
       d.name column_name
  from sys.tables a
       join
       sys.foreign_keys b
    on ( a.name          = 'EMP'
         and a.object_id = b.parent_object_id
       )
       join
       sys.foreign_key_columns c
   on (  b.object_id = c.constraint_object_id )
      join
      sys.columns d
   on (    c.constraint_column_id = d.column_id
       and a.object_id            = d.object_id
      )
      ) fkeys
      left join
      (
select a.name index_name,
      b.object_id,
      b.column_id
  from sys.indexes a,
      sys.index_columns b
```

```
33  where a.index_id = b.index_id
34        ) ind_cols
35     on (     fkeys.object_id = ind_cols.object_id
36          and fkeys.column_id = ind_cols.column_id )
37  where ind_cols.index_name is null
```

Discussion

Each vendor uses its own locking mechanism when modifying rows. In cases where there is a parent-child relationship enforced via foreign key, having indexes on the child column(s) can reducing locking (see your specific RDBMS documentation for details). In other cases, it is common that a child table is joined to a parent table on the foreign key column, so an index may help improve performance in that scenario as well.

5.6 Using SQL to Generate SQL

Problem

You want to create dynamic SQL statements, perhaps to automate maintenance tasks. You want to accomplish three tasks in particular: count the number of rows in your tables, disable foreign key constraints defined on your tables, and generate insert scripts from the data in your tables.

Solution

The concept is to use strings to build SQL statements, and the values that need to be filled in (such as the object name the command acts upon) will be supplied by data from the tables you are selecting from. Keep in mind, the queries only generate the statements; you must then run these statements via script, manually, or however you execute your SQL statements. The following examples are queries that would work on an Oracle system. For other RDBMSs the technique is exactly the same, the only difference being things like the names of the data dictionary tables and date formatting. The output shown from the queries that follow are a portion of the rows returned from an instance of Oracle on my laptop. Your result sets will of course vary:

```
/* generate SQL to count all the rows in all your tables */

select 'select count(*) from '||table_name||';' cnts
  from user_tables;

CNTS
---------------------------------------
select count(*) from ANT;
select count(*) from BONUS;
select count(*) from DEMO1;
select count(*) from DEMO2;
```

```
select count(*) from DEPT;
select count(*) from DUMMY;
select count(*) from EMP;
select count(*) from EMP_SALES;
select count(*) from EMP_SCORE;
select count(*) from PROFESSOR;
select count(*) from T;
select count(*) from T1;
select count(*) from T2;
select count(*) from T3;
select count(*) from TEACH;
select count(*) from TEST;
select count(*) from TRX_LOG;
select count(*) from X;

/* disable foreign keys from all tables */

select 'alter table '||table_name||
       ' disable constraint '||constraint_name||';' cons
  from user_constraints
 where constraint_type = 'R';

CONS
-----------------------------------------------
alter table ANT disable constraint ANT_FK;
alter table BONUS disable constraint BONUS_FK;
alter table DEMO1 disable constraint DEMO1_FK;
alter table DEMO2 disable constraint DEMO2_FK;
alter table DEPT disable constraint DEPT_FK;
alter table DUMMY disable constraint DUMMY_FK;
alter table EMP disable constraint EMP_FK;
alter table EMP_SALES disable constraint EMP_SALES_FK;
alter table EMP_SCORE disable constraint EMP_SCORE_FK;
alter table PROFESSOR disable constraint PROFESSOR_FK;

/* generate an insert script from some columns in table EMP */

select 'insert into emp(empno,ename,hiredate) '||chr(10)||
       'values( '||empno||','||''''||ename
       ||''',to_date('||''''||hiredate||''') );' inserts
  from emp
 where deptno = 10;

INSERTS
------------------------------------------------
insert into emp(empno,ename,hiredate)
values( 7782,'CLARK',to_date('09-JUN-2006 00:00:00') );

insert into emp(empno,ename,hiredate)
values( 7839,'KING',to_date('17-NOV-2006 00:00:00') );
```

```
insert into emp(empno,ename,hiredate)
values( 7934,'MILLER',to_date('23-JAN-2007 00:00:00') );
```

Discussion

Using SQL to generate SQL is particularly useful for creating portable scripts such as you might use when testing on multiple environments. Additionally, as can be seen by the previous examples, using SQL to generate SQL is useful for performing batch maintenance, and for easily finding out information about multiple objects in one go. Generating SQL with SQL is an extremely simple operation, and the more you experiment with it, the easier it will become. The examples provided should give you a nice base on how to build your own "dynamic" SQL scripts because, quite frankly, there's not much to it. Work on it and you'll get it.

5.7 Describing the Data Dictionary Views in an Oracle Database

Problem

You are using Oracle. You can't remember what data dictionary views are available to you, nor can you remember their column definitions. Worse yet, you do not have convenient access to vendor documentation.

Solution

This is an Oracle-specific recipe. Not only does Oracle maintain a robust set of data dictionary views, but there are also data dictionary views to document the data dictionary views. It's all so wonderfully circular.

Query the view named DICTIONARY to list data dictionary views and their purposes:

```
select table_name, comments
  from dictionary
  order by table_name;

TABLE_NAME                     COMMENTS
------------------------------ --------------------------------------------
ALL_ALL_TABLES                 Description of all object and relational
                               tables accessible to the user

ALL_APPLY                      Details about each apply process that
                               dequeues from the queue visible to the
                               current user
…
```

Query DICT_COLUMNS to describe the columns in a given data dictionary view:

```
select column_name, comments
     from dict_columns
 where table_name = 'ALL_TAB_COLUMNS';

COLUMN_NAME                    COMMENTS
------------------------------ -------------------------------------------
OWNER
TABLE_NAME                     Table, view or cluster name
COLUMN_NAME                    Column name
DATA_TYPE                      Datatype of the column
DATA_TYPE_MOD                  Datatype modifier of the column
DATA_TYPE_OWNER                Owner of the datatype of the column
DATA_LENGTH                    Length of the column in bytes
DATA_PRECISION                 Length: decimal digits (NUMBER) or binary
                               digits (FLOAT)
```

Discussion

Back in the day, when Oracle's documentation set wasn't so freely available on the web, it was incredibly convenient that Oracle made the DICTIONARY and DICT_COLUMNS views available. Knowing just those two views, you could bootstrap to learning about all the other views and then shift to learning about your entire database.

Even today, it's convenient to know about DICTIONARY and DICT_COLUMNS. Often, if you aren't quite certain which view describes a given object type, you can issue a wildcard query to find out. For example, to get a handle on what views might describe tables in your schema:

```
select table_name, comments
  from dictionary
 where table_name LIKE '%TABLE%'
 order by table_name;
```

This query returns all data dictionary view names that include the term TABLE. This approach takes advantage of Oracle's fairly consistent data dictionary view naming conventions. Views describing tables are all likely to contain TABLE in their name. (Sometimes, as in the case of ALL_TAB_COLUMNS, TABLE is abbreviated TAB.)

5.8 Summing Up

Queries on metadata open up a range of possibilities for letting SQL do more of the work than you, and they relieve some of the need to *know* your database. This is especially useful as you deal with more complex databases with similarly complex structures.

CHAPTER 6

Working with Strings

This chapter focuses on string manipulation in SQL. Keep in mind that SQL is not designed to perform complex string manipulation, and you can (and will) find working with strings in SQL to be cumbersome and frustrating at times. Despite SQL's limitations, there are some useful built-in functions provided by the different DBMSs, and we've tried to use them in creative ways. This chapter in particular is representative of the message we tried to convey in the introduction; SQL is the good, the bad, and the ugly. Hopefully you take away from this chapter a better appreciation for what can and can't be done in SQL when working with strings. In many cases you'll be surprised by how easy parsing and transforming strings can be, while at other times you'll be aghast by the kind of SQL that is necessary to accomplish a particular task.

Many of the recipes that follow use the TRANSLATE and REPLACE functions that are now available in all the DBMSs covered in this book, with the exception of MySQL, which only has `replace`. In this last case, it is worth noting early on that you can replicate the effect of TRANSLATE by using nested REPLACE functions.

The first recipe in this chapter is critically important, as it is leveraged by several of the subsequent solutions. In many cases, you'd like to have the ability to traverse a string by moving through it a character at a time. Unfortunately, SQL does not make this easy. Because there is limited loop functionality in SQL, you need to mimic a loop to traverse a string. We call this operation "walking a string" or "walking through a string," and the very first recipe explains the technique. This is a fundamental operation in string parsing when using SQL, and is referenced and used by almost all recipes in this chapter. We strongly suggest becoming comfortable with how the technique works.

6.1 Walking a String

Problem

You want to traverse a string to return each character as a row, but SQL lacks a loop operation. For example, you want to display the ENAME "KING" from table EMP as four rows, where each row contains just characters from KING.

Solution

Use a Cartesian product to generate the number of rows needed to return each character of a string on its own line. Then use your DBMS's built-in string parsing function to extract the characters you are interested in (SQL Server users will use SUBSTRING instead of SUBSTR and DATALENGTH instead of LENGTH):

```
1 select substr(e.ename,iter.pos,1) as C
2   from (select ename from emp where ename = 'KING') e,
3        (select id as pos from t10) iter
4  where iter.pos <= length(e.ename)


C
-
K
I
N
G
```

Discussion

The key to iterating through a string's characters is to join against a table that has enough rows to produce the required number of iterations. This example uses table T10, which contains 10 rows (it has one column, ID, holding the values 1 through 10). The maximum number of rows that can be returned from this query is 10.

The following example shows the Cartesian product between E and ITER (i.e., between the specific name and the 10 rows from T10) without parsing ENAME:

```
select ename, iter.pos
  from (select ename from emp where ename = 'KING') e,
       (select id as pos from t10) iter

ENAME             POS
---------- ----------
KING               1
KING               2
KING               3
KING               4
KING               5
```

```
KING                 6
KING                 7
KING                 8
KING                 9
KING                10
```

The cardinality of inline view E is 1, and the cardinality of inline view ITER is 10. The Cartesian product is then 10 rows. Generating such a product is the first step in mimicking a loop in SQL.

It is common practice to refer to table T10 as a "pivot" table.

The solution uses a WHERE clause to break out of the loop after four rows have been returned. To restrict the result set to the same number of rows as there are characters in the name, that WHERE clause specifies ITER.POS <= LENGTH(E. ENAME) as the condition:

```
select ename, iter.pos
  from (select ename from emp where ename = 'KING') e,
       (select id as pos from t10) iter
 where iter.pos <= length(e.ename)

ENAME             POS
---------- ----------
KING                1
KING                2
KING                3
KING                4
```

Now that you have one row for each character in E.ENAME, you can use ITER.POS as a parameter to SUBSTR, allowing you to navigate through the characters in the string. ITER.POS increments with each row, and thus each row can be made to return a successive character from E.ENAME. This is how the solution example works.

Depending on what you are trying to accomplish, you may or may not need to generate a row for every single character in a string. The following query is an example of walking E.ENAME and exposing different portions (more than a single character) of the string:

```
select substr(e.ename,iter.pos) a,
       substr(e.ename,length(e.ename)-iter.pos+1) b
  from (select ename from emp where ename = 'KING') e,
       (select id pos from t10) iter
 where iter.pos <= length(e.ename)
```

```
A          B
---------- ----------
KING       G
ING        NG
NG         ING
G          KING
```

The most common scenarios for the recipes in this chapter involve walking the whole string to generate a row for each character in the string, or walking the string such that the number of rows generated reflects the number of particular characters or delimiters that are present in the string.

6.2 Embedding Quotes Within String Literals

Problem

You want to embed quote marks within string literals. You would like to produce results such as the following with SQL:

```
QMARKS
--------------
g'day mate
beavers' teeth
'
```

Solution

The following three SELECTs highlight different ways you can create quotes: in the middle of a string and by themselves:

```
1 select 'g''day mate' qmarks from t1 union all
2 select 'beavers'' teeth'    from t1 union all
3 select ''''                 from t1
```

Discussion

When working with quotes, it's often useful to think of them like parentheses. When you have an opening parenthesis, you must always have a closing parenthesis. The same goes for quotes. Keep in mind that you should always have an even number of quotes across any given string. To embed a single quote within a string, you need to use two quotes:

```
select 'apples core', 'apple''s core',
       case when '' is null then 0 else 1 end
  from t1

 'APPLESCORE 'APPLE''SCOR CASEWHEN''ISNULLTHEN0ELSE1END
 ----------- ------------ ----------------------------
 apples core apple's core                            0
```

The following is the solution stripped down to its bare elements. You have two outer quotes defining a string literal, and within that string literal, you have two quotes that together represent just one quote in the string that you actually get:

```
select '''' as quote from t1

Q
-
'
```

When working with quotes, be sure to remember that a string literal comprising two quotes alone, with no intervening characters, is NULL.

6.3 Counting the Occurrences of a Character in a String

Problem

You want to count the number of times a character or substring occurs within a given string. Consider the following string:

```
10,CLARK,MANAGER
```

You want to determine how many commas are in the string.

Solution

Subtract the length of the string without the commas from the original length of the string to determine the number of commas in the string. Each DBMS provides functions for obtaining the length of a string and removing characters from a string. In most cases, these functions are LENGTH and REPLACE, respectively (SQL Server users will use the built-in function LEN rather than LENGTH):

```
1 select (length('10,CLARK,MANAGER')-
2        length(replace('10,CLARK,MANAGER',',','')))/length(',')
3        as cnt
4   from t1
```

Discussion

You arrive at the solution by using simple subtraction. The call to LENGTH on line 1 returns the original size of the string, and the first call to LENGTH on line 2 returns the size of the string without the commas, which are removed by REPLACE.

By subtracting the two lengths, you obtain the difference in terms of characters, which is the number of commas in the string. The last operation divides the difference by the length of your search string. This division is necessary if the string you are looking for has a length greater than 1. In the following example, counting the

occurrence of "LL" in the string "HELLO HELLO" without dividing will return an incorrect result:

```
select
       (length('HELLO HELLO')-
       length(replace('HELLO HELLO','LL','')))/length('LL')
       as correct_cnt,
       (length('HELLO HELLO')-
       length(replace('HELLO HELLO','LL',''))) as incorrect_cnt
  from t1

CORRECT_CNT INCORRECT_CNT
----------- -------------
          2             4
```

6.4 Removing Unwanted Characters from a String

Problem

You want to remove specific characters from your data. A scenario where this may occur is in dealing with badly formatted numeric data, especially currency data, where commas have been used to separate zeros, and currency markers are mixed in the column with the quantity. Another scenario is that you want to export data from your database as a CSV file, but there is a text field containing commas, which will be read as separators when the CSV file is accessed. Consider this result set:

```
ENAME             SAL
---------- ----------
SMITH             800
ALLEN            1600
WARD             1250
JONES            2975
MARTIN           1250
BLAKE            2850
CLARK            2450
SCOTT            3000
KING             5000
TURNER           1500
ADAMS            1100
JAMES             950
FORD             3000
MILLER           1300
```

You want to remove all zeros and vowels as shown by the following values in columns STRIPPED1 and STRIPPED2:

```
ENAME      STRIPPED1         SAL STRIPPED2
---------- ---------- ---------- ---------
SMITH      SMTH              800 8
ALLEN      LLN              1600 16
```

```
WARD       WRD              1250 125
JONES      JNS              2975 2975
MARTIN     MRTN             1250 125
BLAKE      BLK              2850 285
CLARK      CLRK             2450 245
SCOTT      SCTT             3000 3
KING       KNG              5000 5
TURNER     TRNR             1500 15
ADAMS      DMS              1100 11
JAMES      JMS               950 95
FORD       FRD              3000 3
MILLER     MLLR             1300 13
```

Solution

Each DBMS provides functions for removing unwanted characters from a string. The functions REPLACE and TRANSLATE are most useful for this problem.

DB2, Oracle, PostgreSQL, and SQL Server

Use the built-in functions TRANSLATE and REPLACE to remove unwanted characters and strings:

```
1 select ename,
2        replace(translate(ename,'aaaaa','AEIOU'),'a','') as stripped1,
3        sal,
4        replace(cast(sal as char(4)),'0','') as stripped2
5   from emp
```

Note that for DB2, the AS keyword is optional for assigning a column alias and can be left out.

MySQL

MySQL does not offer a TRANSLATE function, so several calls to REPLACE are needed:

```
 1 select ename,
 2        replace(
 3        replace(
 4        replace(
 5        replace(
 6        replace(ename,'A',''),'E',''),'I',''),'O',''),'U','')
 7        as stripped1,
 8        sal,
 9        replace(sal,0,'') stripped2
10   from emp
```

Discussion

The built-in function REPLACE removes all occurrences of zeros. To remove the vowels, use TRANSLATE to convert all vowels into one specific character (we used "a"; you can use any character); then use REPLACE to remove all occurrences of that character.

6.5 Separating Numeric and Character Data

Problem

You have numeric data stored with character data together in one column. This could easily happen if you inherit data where units of measurement or currency have been stored with their quantity (e.g., a column with *100 km*, *AUD$200*, or *40 pounds*, rather than either the column making the units clear or a separate column showing the units where necessary).

You want to separate the character data from the numeric data. Consider the following result set:

```
DATA
---------------
SMITH800
ALLEN1600
WARD1250
JONES2975
MARTIN1250
BLAKE2850
CLARK2450
SCOTT3000
KING5000
TURNER1500
ADAMS1100
JAMES950
FORD3000
MILLER1300
```

You would like the result to be:

```
ENAME             SAL
---------- ----------
SMITH             800
ALLEN            1600
WARD             1250
JONES            2975
MARTIN           1250
BLAKE            2850
CLARK            2450
SCOTT            3000
KING             5000
```

```
TURNER          1500
ADAMS           1100
JAMES            950
FORD            3000
MILLER          1300
```

Solution

Use the built-in functions TRANSLATE and REPLACE to isolate the character from the numeric data. Like other recipes in this chapter, the trick is to use TRANSLATE to transform multiple characters into a single character you can reference. This way you are no longer searching for multiple numbers or characters; rather, you are searching for just one character to represent all numbers or one character to represent all characters.

DB2

Use the functions TRANSLATE and REPLACE to isolate and separate the numeric from the character data:

```
 1 select replace(
 2      translate(data,'0000000000','0123456789'),'0','') ename,
 3        cast(
 4      replace(
 5    translate(lower(data),repeat('z',26),
 6           'abcdefghijklmnopqrstuvwxyz'),'z','') as integer) sal
 7    from (
 8  select ename||cast(sal as char(4)) data
 9    from emp
10         ) x
```

Oracle

Use the functions TRANSLATE and REPLACE to isolate and separate the numeric from the character data:

```
 1 select replace(
 2      translate(data,'0123456789','0000000000'),'0') ename,
 3      to_number(
 4        replace(
 5        translate(lower(data),
 6                  'abcdefghijklmnopqrstuvwxyz',
 7                   rpad('z',26,'z')),'z')) sal
 8   from (
 9 select ename||sal data
10   from emp
11        )
```

PostgreSQL

Use the functions TRANSLATE and REPLACE to isolate and separate the numeric from the character data:

```
select replace(
     translate(data,'0123456789','0000000000'),'0','') as ename,
         cast(
       replace(
     translate(lower(data),
               'abcdefghijklmnopqrstuvwxyz',
               rpad('z',26,'z')),'z','') as integer) as sal
  from (
select ename||sal as data
  from emp
       ) x
```

SQL Server

Use the functions TRANSLATE and REPLACE to isolate and separate the numeric from the character data:

```
select replace(
     translate(data,'0123456789','0000000000'),'0','') as ename,
         cast(
       replace(
     translate(lower(data),
               'abcdefghijklmnopqrstuvwxyz',
               replicate('z',26),'z','') as integer) as sal
  from (
select concat(ename,sal) as data
  from emp
       ) x
```

Discussion

The syntax is a bit different for each DBMS, but the technique is the same. The syntax is slightly different for each DBMS, but the technique is the same; we will use the Oracle solution for this discussion. The key to solving this problem is to isolate the numeric and character data. You can use TRANSLATE and REPLACE to do this. To extract the numeric data, first isolate all character data using TRANSLATE:

```
select data,
       translate(lower(data),
                 'abcdefghijklmnopqrstuvwxyz',
                 rpad('z',26,'z')) sal
  from (select ename||sal data from emp)

DATA                 SAL
-------------------- -------------------
SMITH800             zzzzz800
ALLEN1600            zzzzz1600
```

```
WARD1250             zzzz1250
JONES2975            zzzzz2975
MARTIN1250           zzzzzz1250
BLAKE2850            zzzzz2850
CLARK2450            zzzzz2450
SCOTT3000            zzzzz3000
KING5000             zzzz5000
TURNER1500           zzzzzz1500
ADAMS1100            zzzzz1100
JAMES950             zzzzz950
FORD3000             zzzz3000
MILLER1300           zzzzzz1300
```

By using TRANSLATE you convert every nonnumeric character into a lowercase Z. The next step is to remove all instances of lowercase Z from each record using REPLACE, leaving only numerical characters that can then be cast to a number:

```
select data,
       to_number(
         replace(
       translate(lower(data),
                 'abcdefghijklmnopqrstuvwxyz',
                 rpad('z',26,'z')),'z')) sal
  from (select ename||sal data from emp)

DATA                       SAL
-------------------- ----------
SMITH800                   800
ALLEN1600                 1600
WARD1250                  1250
JONES2975                 2975
MARTIN1250                1250
BLAKE2850                 2850
CLARK2450                 2450
SCOTT3000                 3000
KING5000                  5000
TURNER1500                1500
ADAMS1100                 1100
JAMES950                   950
FORD3000                  3000
MILLER1300                1300
```

To extract the nonnumeric characters, isolate the numeric characters using TRANSLATE:

```
select data,
       translate(data,'0123456789','0000000000') ename
  from (select ename||sal data from emp)

DATA                 ENAME
-------------------- ----------
SMITH800             SMITH000
ALLEN1600            ALLEN0000
```

```
WARD1250            WARD0000
JONES2975           JONES0000
MARTIN1250          MARTIN0000
BLAKE2850           BLAKE0000
CLARK2450           CLARK0000
SCOTT3000           SCOTT0000
KING5000            KING0000
TURNER1500          TURNER0000
ADAMS1100           ADAMS0000
JAMES950            JAMES000
FORD3000            FORD0000
MILLER1300          MILLER0000
```

By using TRANSLATE, you convert every numeric character into a zero. The next step is to remove all instances of zero from each record using REPLACE, leaving only nonnumeric characters:

```
select data,
       replace(translate(data,'0123456789','0000000000'),'0') ename
  from (select ename||sal data from emp)

DATA                ENAME
------------------- -------
SMITH800            SMITH
ALLEN1600           ALLEN
WARD1250            WARD
JONES2975           JONES
MARTIN1250          MARTIN
BLAKE2850           BLAKE
CLARK2450           CLARK
SCOTT3000           SCOTT
KING5000            KING
TURNER1500          TURNER
ADAMS1100           ADAMS
JAMES950            JAMES
FORD3000            FORD
MILLER1300          MILLER
```

Put the two techniques together and you have your solution.

6.6 Determining Whether a String Is Alphanumeric

Problem

You want to return rows from a table only when a column of interest contains no characters other than numbers and letters. Consider the following view V (SQL Server users will use the operator + for concatenation instead of ||):

```
create view V as
select ename as data
  from emp
 where deptno=10
 union all
select ename||', $'|| cast(sal as char(4)) ||'.00' as data
  from emp
 where deptno=20
 union all
select ename|| cast(deptno as char(4)) as data
  from emp
 where deptno=30
```

The view V represents your table, and it returns the following:

```
DATA
--------------------
CLARK
KING
MILLER
SMITH, $800.00
JONES, $2975.00
SCOTT, $3000.00
ADAMS, $1100.00
FORD, $3000.00
ALLEN30
WARD30
MARTIN30
BLAKE30
TURNER30
JAMES30
```

However, from the view's data you want to return only the following records:

```
DATA
-------------
CLARK
KING
MILLER
ALLEN30
WARD30
MARTIN30
BLAKE30
TURNER30
JAMES30
```

In short, you want to omit those rows containing data other than letters and digits.

Solution

It may seem intuitive at first to solve the problem by searching for all the possible non-alphanumeric characters that can be found in a string, but, on the contrary, you will find it easier to do the exact opposite: find all the alphanumeric characters. By

doing so, you can treat all the alphanumeric characters as one by converting them to one single character. The reason you want to do this is so the alphanumeric characters can be manipulated together, as a whole. Once you've generated a copy of the string in which all alphanumeric characters are represented by a single character of your choosing, it is easy to isolate the alphanumeric characters from any other characters.

DB2

Use the function TRANSLATE to convert all alphanumeric characters to a single character; then identify any rows that have characters other than the converted alphanumeric character. For DB2 users, the CAST function calls in view V are necessary; otherwise, the view cannot be created due to type conversion errors. Take extra care when working with casts to CHAR as they are fixed length (padded):

```
1 select data
2   from V
3  where translate(lower(data),
4                  repeat('a',36),
5                  '0123456789abcdefghijklmnopqrstuvwxyz') =
6                  repeat('a',length(data))
```

MySQL

The syntax for view V is slightly different in MySQL:

```
create view V as
select ename as data
  from emp
 where deptno=10
 union all
select concat(ename,', $',sal,'.00') as data
  from emp
 where deptno=20
 union all
select concat(ename,deptno) as data
  from emp
 where deptno=30
```

Use a regular expression to easily find rows that contain non-alphanumeric data:

```
1 select data
2   from V
3  where data regexp '[^0-9a-zA-Z]' = 0
```

Oracle and PostgreSQL

Use the function TRANSLATE to convert all alphanumeric characters to a single character; then identify any rows that have characters other than the converted alphanumeric character. The CAST function calls in view V are not needed for Oracle and

PostgreSQL. Take extra care when working with casts to CHAR as they are fixed length (padded).

If you decide to cast, cast to VARCHAR or VARCHAR2:

```
1 select data
2   from V
3  where translate(lower(data),
4                  '0123456789abcdefghijklmnopqrstuvwxyz',
5                  rpad('a',36,'a')) = rpad('a',length(data),'a')
```

SQL Server

The technique is the same, with the exception of there being no RPAD in SQL Server:

```
1 select data
2   from V
3  where translate(lower(data),
4                  '0123456789abcdefghijklmnopqrstuvwxyz',
5                  replicate('a',36)) = replicate('a',len(data))
```

Discussion

The key to these solutions is being able to reference multiple characters concurrently. By using the function TRANSLATE, you can easily manipulate all numbers or all characters without having to "iterate" and inspect each character one by one.

DB2, Oracle, PostgreSQL, and SQL Server

Only 9 of the 14 rows from view V are alphanumeric. To find the rows that are alphanumeric only, simply use the function TRANSLATE. In this example, TRANSLATE converts characters 0–9 and a–z to "a". Once the conversion is done, the converted row is then compared with a string of all "a" with the same length (as the row). If the length is the same, then you know all the characters are alphanumeric and nothing else.

By using the TRANSLATE function (using the Oracle syntax):

```
where translate(lower(data),
                '0123456789abcdefghijklmnopqrstuvwxyz',
                 rpad('a',36,'a'))
```

you convert all numbers and letters into a distinct character (we chose "a"). Once the data is converted, all strings that are indeed alphanumeric can be identified as a string comprising only a single character (in this case, "a"). This can be seen by running TRANSLATE by itself:

```
select data, translate(lower(data),
                  '0123456789abcdefghijklmnopqrstuvwxyz',
                   rpad('a',36,'a'))
  from V
```

```
DATA                 TRANSLATE(LOWER(DATA)
-------------------- ---------------------
CLARK                aaaaa
…
SMITH, $800.00       aaaaa, $aaa.aa
…
ALLEN30              aaaaaaa
…
```

The alphanumeric values are converted, but the string lengths have not been modified. Because the lengths are the same, the rows to keep are the ones for which the call to TRANSLATE returns all "a"s. You keep those rows, rejecting the others, by comparing each original string's length with the length of its corresponding string of "a"s:

```
select data, translate(lower(data),
                  '0123456789abcdefghijklmnopqrstuvwxyz',
                   rpad('a',36,'a')) translated,
        rpad('a',length(data),'a') fixed
  from V

DATA                 TRANSLATED           FIXED
-------------------- -------------------- ----------------
CLARK                aaaaa                aaaaa
…
SMITH, $800.00       aaaaa, $aaa.aa       aaaaaaaaaaaaaa
…
ALLEN30              aaaaaaa              aaaaaaa
…
```

The last step is to keep only the strings where TRANSLATED equals FIXED.

MySQL

The expression in the WHERE clause:

```
where data regexp '[^0-9a-zA-Z]' = 0
```

causes rows that have only numbers or characters to be returned. The value ranges in the brackets, "0-9a-zA-Z", represent all possible numbers and letters. The character ^ is for negation, so the expression can be stated as "not numbers or letters." A return value of 1 is true and 0 is false, so the whole expression can be stated as "return rows where anything other than numbers and letters is false."

6.7 Extracting Initials from a Name

Problem

You want convert a full name into initials. Consider the following name:

```
Stewie Griffin
```

You would like to return:

```
S.G.
```

Solution

It's important to keep in mind that SQL does not provide the flexibility of languages such as C or Python; therefore, creating a generic solution to deal with any name format is not something particularly easy to do in SQL. The solutions presented here expect the names to be either first and last name, or first, middle name/middle initial, and last name.

DB2

Use the built-in functions REPLACE, TRANSLATE, and REPEAT to extract the initials:

```
select replace(
       replace(
       translate(replace('Stewie Griffin', '.', ''),
                 repeat('#',26),
                 'abcdefghijklmnopqrstuvwxyz'),
                  '#','' ), ' ','.' )
                 ||'.'
  from t1
```

MySQL

Use the built-in functions CONCAT, CONCAT_WS, SUBSTRING, and SUBSTRING_ INDEX to extract the initials:

```
select case
         when cnt = 2 then
           trim(trailing '.' from
                concat_ws('.',
                 substr(substring_index(name,' ',1),1,1),
                 substr(name,
                        length(substring_index(name,' ',1))+2,1),
                 substr(substring_index(name,' ',-1),1,1),
                 '.'))
         else
           trim(trailing '.' from
                concat_ws('.',
                 substr(substring_index(name,' ',1),1,1),
                 substr(substring_index(name,' ',-1),1,1)
                 ))
         end as initials
  from (
select name,length(name)-length(replace(name,' ','')) as cnt
  from (
select replace('Stewie Griffin','.','') as name from t1
```

```
21        )y
22        )x
```

Oracle and PostgreSQL

Use the built-in functions REPLACE, TRANSLATE, and RPAD to extract the initials:

```
1 select replace(
2        replace(
3        translate(replace('Stewie Griffin', '.', ''),
4                  'abcdefghijklmnopqrstuvwxyz',
5                  rpad('#',26,'#') ), '#','' ),' ','.' ) ||'.'
6   from t1
```

SQL Server

```
1 select replace(
2        replace(
3        translate(replace('Stewie Griffin', '.', ''),
4                  'abcdefghijklmnopqrstuvwxyz',
5                  replicate('#',26) ), '#','' ),' ','.' ) + '.'
6   from t1
```

Discussion

By isolating the capital letters, you can extract the initials from a name. The following sections describe each vendor-specific solution in detail.

DB2

The REPLACE function will remove any periods in the name (to handle middle initials), and the TRANSLATE function will convert all non-uppercase letters to #.

```
select translate(replace('Stewie Griffin', '.', ''),
                 repeat('#',26),
                 'abcdefghijklmnopqrstuvwxyz')
  from t1

TRANSLATE('STE
--------------
S##### G######
```

At this point, the initials are the characters that are not #. The function REPLACE is then used to remove all the # characters:

```
select replace(
       translate(replace('Stewie Griffin', '.', ''),
                  repeat('#',26),
                  'abcdefghijklmnopqrstuvwxyz'),'#','')
  from t1
```

```
REP
---
S G
```

The next step is to replace the white space with a period by using REPLACE again:

```
select replace(
               replace(
               translate(replace('Stewie Griffin', '.', ''),
                         repeat('#',26),
                        'abcdefghijklmnopqrstuvwxyz'),'#',''),' ','.') || '.'
         from t1

        REPLA
        -----
        S.G
```

The final step is to append a decimal to the end of the initials.

Oracle and PostgreSQL

The REPLACE function will remove any periods in the name (to handle middle initials), and the TRANSLATE function will convert all non-uppercase letters to #.

```
select translate(replace('Stewie Griffin','.',''),
                 'abcdefghijklmnopqrstuvwxyz',
                 rpad('#',26,'#'))
  from t1

TRANSLATE('STE
--------------
S##### G######
```

At this point, the initials are the characters that are not #. The function REPLACE is then used to remove all the # characters:

```
select replace(
       translate(replace('Stewie Griffin','.',''),
                 'abcdefghijklmnopqrstuvwxyz',
                  rpad('#',26,'#')),'#','')
  from t1

REP
---
S G
```

The next step is to replace the white space with a period by using REPLACE again:

```
select replace(
       replace(
     translate(replace('Stewie Griffin','.',''),
               'abcdefghijklmnopqrstuvwxyz',
               rpad('#',26,'#') ),'#',''),' ','.') || '.'
  from t1
```

```
REPLA
-----
S.G
```

The final step is to append a decimal to the end of the initials.

MySQL

The inline view Y is used to remove any period from the name. The inline view X finds the number of white spaces in the name so the SUBSTR function can be called the correct number of times to extract the initials. The three calls to SUBSTRING_INDEX parse the string into individual names based on the location of the white space. Because there is only a first and last name, the code in the ELSE portion of the case statement is executed:

```
select substr(substring_index(name, ' ',1),1,1) as a,
       substr(substring_index(name,' ',-1),1,1) as b
  from (select 'Stewie Griffin' as name from t1) x

A B
- -
S G
```

If the name in question has a middle name or initial, the initial would be returned by executing:

```
substr(name,length(substring_index(name, ' ',1))+2,1)
```

which finds the end of the first name and then moves two spaces to the beginning of the middle name or initial, that is, the start position for SUBSTR. Because only one character is kept, the middle name or initial is successfully returned. The initials are then passed to CONCAT_WS, which separates the initials by a period:

```
select concat_ws('.',
                 substr(substring_index(name, ' ',1),1,1),
                 substr(substring_index(name,' ',-1),1,1),
                 '.' ) a
  from (select 'Stewie Griffin' as name from t1) x

A
-----
S.G..
```

The last step is to trim the extraneous period from the initials.

6.8 Ordering by Parts of a String

Problem

You want to order your result set based on a substring. Consider the following records:

```
ENAME
----------
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
ADAMS
JAMES
FORD
MILLER
```

You want the records to be ordered based on the *last* two characters of each name:

```
ENAME
---------
ALLEN
TURNER
MILLER
JONES
JAMES
MARTIN
BLAKE
ADAMS
KING
WARD
FORD
CLARK
SMITH
SCOTT
```

Solution

The key to this solution is to find and use your DBMS's built-in function to extract the substring on which you want to sort. This is typically done with the SUBSTR function.

DB2, Oracle, MySQL, and PostgreSQL

Use a combination of the built-in functions LENGTH and SUBSTR to order by a specific part of a string:

```
1 select ename
2   from emp
3  order by substr(ename,length(ename)-1,)
```

SQL Server

Use functions SUBSTRING and LEN to order by a specific part of a string:

```
1 select ename
2   from emp
3  order by substring(ename,len(ename)-1,2)
```

Discussion

By using a SUBSTR expression in your ORDER BY clause, you can pick any part of a string to use in ordering a result set. You're not limited to SUBSTR either. You can order rows by the result of almost any expression.

6.9 Ordering by a Number in a String

Problem

You want order your result set based on a number within a string. Consider the following view:

```
create view V as
select e.ename ||' '||
        cast(e.empno as char(4))||' '||
        d.dname as data
  from emp e, dept d
 where e.deptno=d.deptno
```

This view returns the following data:

```
DATA
 ----------------------------
 CLARK   7782 ACCOUNTING
 KING    7839 ACCOUNTING
 MILLER  7934 ACCOUNTING
 SMITH   7369 RESEARCH
 JONES   7566 RESEARCH
 SCOTT   7788 RESEARCH
 ADAMS   7876 RESEARCH
 FORD    7902 RESEARCH
 ALLEN   7499 SALES
 WARD    7521 SALES
```

```
 MARTIN  7654 SALES
 BLAKE   7698 SALES
 TURNER  7844 SALES
 JAMES   7900 SALES
```

You want to order the results based on the employee number, which falls between the employee name and respective department:

```
DATA
---------------------------
SMITH    7369 RESEARCH
ALLEN    7499 SALES
WARD     7521 SALES
JONES    7566 RESEARCH
MARTIN   7654 SALES
BLAKE    7698 SALES
CLARK    7782 ACCOUNTING
SCOTT    7788 RESEARCH
KING     7839 ACCOUNTING
TURNER   7844 SALES
ADAMS    7876 RESEARCH
JAMES    7900 SALES
FORD     7902 RESEARCH
MILLER   7934 ACCOUNTING
```

Solution

Each solution uses functions and syntax specific to its DBMS, but the method (making use of the built-in functions REPLACE and TRANSLATE) is the same for each. The idea is to use REPLACE and TRANSLATE to remove nondigits from the strings, leaving only the numeric values upon which to sort.

DB2

Use the built-in functions REPLACE and TRANSLATE to order by numeric characters in a string:

```
1 select data
2   from V
3  order by
4         cast(
5      replace(
6    translate(data,repeat('#',length(data)),
7      replace(
8    translate(data,'##########','0123456789'),
9             '#','')),'#','') as integer)
```

Oracle

Use the built-in functions REPLACE and TRANSLATE to order by numeric characters in a string:

```
1 select data
2   from V
3  order by
4         to_number(
5           replace(
6         translate(data,
7           replace(
8         translate(data,'0123456789','##########'),
9                  '#'),rpad('#',20,'#')),'#'))
```

PostgreSQL

Use the built-in functions REPLACE and TRANSLATE to order by numeric characters in a string:

```
1 select data
2   from V
3  order by
4         cast(
5      replace(
6    translate(data,
7      replace(
8    translate(data,'0123456789','##########'),
9             '#',''),rpad('#',20,'#')),'#','') as integer)
```

MySQL

As of the time of this writing, MySQL does not provide the TRANSLATE function.

Discussion

The purpose of view V is only to supply rows on which to demonstrate this recipe's solution. The view simply concatenates several columns from the EMP table. The solution shows how to take such concatenated text as input and sort it by the employee number embedded within.

The ORDER BY clause in each solution may look intimidating, but it performs quite well and is straightforward once you examine it piece by piece. To order by the numbers in the string, it's easiest to remove any characters that are not numbers. Once the nonnumeric characters are removed, all that is left to do is cast the string of numerals into a number and then sort as you see fit. Before examining each function call, it is important to understand the order in which each function is called. Starting with the innermost call, TRANSLATE (line 8 from each of the original solutions), you see that:

From the innermost call, the sequence of steps is TRANSLATE (line 8); REPLACE (line 7) ; TRANSLATE (line 6); REPLACE (line 5). The final step is to use CAST to return the result as a number.

The first step is to convert the numbers into characters that do not exist in the rest of the string. For this example, we chose # and used TRANSLATE to convert all nonnumeric characters into occurrences of #. For example, the following query shows the original data on the left and the results from the first translation:

```
select data,
       translate(data,'0123456789','##########') as tmp
  from V

DATA                           TMP
------------------------------ ----------------------
CLARK   7782 ACCOUNTING        CLARK   #### ACCOUNTING
KING    7839 ACCOUNTING        KING    #### ACCOUNTING
MILLER  7934 ACCOUNTING        MILLER  #### ACCOUNTING
SMITH   7369 RESEARCH          SMITH   #### RESEARCH
JONES   7566 RESEARCH          JONES   #### RESEARCH
SCOTT   7788 RESEARCH          SCOTT   #### RESEARCH
ADAMS   7876 RESEARCH          ADAMS   #### RESEARCH
FORD    7902 RESEARCH          FORD    #### RESEARCH
ALLEN   7499 SALES             ALLEN   #### SALES
WARD    7521 SALES             WARD    #### SALES
MARTIN  7654 SALES             MARTIN  #### SALES
BLAKE   7698 SALES             BLAKE   #### SALES
TURNER 7844 SALES              TURNER  #### SALES
JAMES  7900 SALES              JAMES   #### SALES
```

TRANSLATE finds the numerals in each string and converts each one to the # character. The modified strings are then returned to REPLACE (line 11), which removes all occurrences of #:

```
select data,
replace(
translate(data,'0123456789','##########'),'#') as tmp
  from V

DATA                           TMP
------------------------------ -------------------
CLARK   7782 ACCOUNTING        CLARK   ACCOUNTING
KING    7839 ACCOUNTING        KING    ACCOUNTING
MILLER  7934 ACCOUNTING        MILLER  ACCOUNTING
SMITH   7369 RESEARCH          SMITH   RESEARCH
JONES   7566 RESEARCH          JONES   RESEARCH
SCOTT   7788 RESEARCH          SCOTT   RESEARCH
ADAMS   7876 RESEARCH          ADAMS   RESEARCH
FORD    7902 RESEARCH          FORD    RESEARCH
ALLEN   7499 SALES             ALLEN   SALES
WARD    7521 SALES             WARD    SALES
MARTIN  7654 SALES             MARTIN  SALES
BLAKE   7698 SALES             BLAKE   SALES
TURNER  7844 SALES             TURNER  SALES
JAMES   7900 SALES             JAMES   SALES
```

The strings are then returned to TRANSLATE once again, but this time it's the second (outermost) TRANSLATE in the solution. TRANSLATE searches the original string for any characters that match the characters in TMP. If any are found, they too are converted to #s.

This conversion allows all nonnumeric characters to be treated as a single character (because they are all transformed to the same character):

```
select data, translate(data,
             replace(
             translate(data,'0123456789','##########'),
             '#'),
             rpad('#',length(data),'#')) as tmp
  from V

DATA                           TMP
------------------------------ ---------------------------
CLARK   7782 ACCOUNTING        #######7782###########
KING    7839 ACCOUNTING        #######7839###########
MILLER  7934 ACCOUNTING        #######7934###########
SMITH   7369 RESEARCH          #######7369#########
JONES   7566 RESEARCH          #######7566#########
SCOTT   7788 RESEARCH          #######7788#########
ADAMS   7876 RESEARCH          #######7876#########
FORD    7902 RESEARCH          #######7902#########
ALLEN   7499 SALES             #######7499######
WARD    7521 SALES             #######7521######
MARTIN  7654 SALES             #######7654######
BLAKE   7698 SALES             #######7698######
TURNER  7844 SALES             #######7844######
JAMES   7900 SALES             #######7900######
```

The next step is to remove all # characters through a call to REPLACE (line 8), leaving you with only numbers:

```
select data, replace(
             translate(data,
             replace(
           translate(data,'0123456789','##########'),
                     '#'),
                     rpad('#',length(data),'#')),'#') as tmp
  from V

DATA                           TMP
------------------------------ -----------
CLARK   7782 ACCOUNTING        7782
KING    7839 ACCOUNTING        7839
MILLER  7934 ACCOUNTING        7934
SMITH   7369 RESEARCH          7369
JONES   7566 RESEARCH          7566
SCOTT   7788 RESEARCH          7788
ADAMS   7876 RESEARCH          7876
```

```
FORD    7902 RESEARCH          7902
ALLEN   7499 SALES             7499
WARD    7521 SALES             7521
MARTIN  7654 SALES             7654
BLAKE   7698 SALES             7698
TURNER  7844 SALES             7844
JAMES   7900 SALES             7900
```

Finally, cast TMP to a number (line 4) using the appropriate DBMS function (often CAST) to accomplish this:

```
select data, to_number(
              replace(
             translate(data,
             replace(
       translate(data,'0123456789','##########'),
                  '#'),
                  rpad('#',length(data),'#')),'#')) as tmp
  from V

DATA                                 TMP
------------------------------ ----------
CLARK   7782 ACCOUNTING              7782
KING    7839 ACCOUNTING              7839
MILLER  7934 ACCOUNTING              7934
SMITH   7369 RESEARCH                7369
JONES   7566 RESEARCH                7566
SCOTT   7788 RESEARCH                7788
ADAMS   7876 RESEARCH                7876
FORD    7902 RESEARCH                7902
ALLEN   7499 SALES                   7499
WARD    7521 SALES                   7521
MARTIN  7654 SALES                   7654
BLAKE   7698 SALES                   7698
TURNER  7844 SALES                   7844
JAMES   7900 SALES                   7900
```

When developing queries like this, it's helpful to work with your expressions in the SELECT list. That way, you can easily view the intermediate results as you work toward a final solution. However, because the point of this recipe is to order the results, ultimately you should place all the function calls into the ORDER BY clause:

```
select data
  from V
 order by
        to_number(
          replace(
        translate( data,
          replace(
        translate( data,'0123456789','##########'),
                  '#'),rpad('#',length(data),'#')),'#'))
```

```
DATA
---------------------------
SMITH   7369 RESEARCH
ALLEN   7499 SALES
WARD    7521 SALES
JONES   7566 RESEARCH
MARTIN  7654 SALES
BLAKE   7698 SALES
CLARK   7782 ACCOUNTING
SCOTT   7788 RESEARCH
KING    7839 ACCOUNTING
TURNER  7844 SALES
ADAMS   7876 RESEARCH
JAMES   7900 SALES
FORD    7902 RESEARCH
MILLER  7934 ACCOUNTING
```

As a final note, the data in the view is comprised of three fields, only one being numeric. Keep in mind that if there had been multiple numeric fields, they would have all been concatenated into one number before the rows were sorted.

6.10 Creating a Delimited List from Table Rows

Problem

You want to return table rows as values in a delimited list, perhaps delimited by commas, rather than in vertical columns as they normally appear. You want to convert a result set from this:

```
DEPTNO EMPS
------ ----------
    10 CLARK
    10 KING
    10 MILLER
    20 SMITH
    20 ADAMS
    20 FORD
    20 SCOTT
    20 JONES
    30 ALLEN
    30 BLAKE
    30 MARTIN
    30 JAMES
    30 TURNER
    30 WARD
```

to this:

```
 DEPTNO EMPS
------- -----------------------------------
     10 CLARK,KING,MILLER
```

```
20 SMITH,JONES,SCOTT,ADAMS,FORD
30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES
```

Solution

Each DBMS requires a different approach to this problem. The key is to take advantage of the built-in functions provided by your DBMS. Understanding what is available to you will allow you to exploit your DBMS's functionality and come up with creative solutions for a problem that is typically not solved in SQL.

Most DBMSs have now adopted a function specifically designed to concatenate strings, such as MySQL's GROUP_CONCAT function (one of the earliest) or STRING_ADD (added to SQL Server as recently as SQL Server 2017). These functions have similar syntax, and make this task straightforward.

DB2

Use LIST_AGG to build the delimited list:

```
1 select deptno,
2        list_agg(ename ',') within GROUP(Order by 0) as emps
3   from emp
4  group by deptno
```

MySQL

Use the built-in function GROUP_CONCAT to build the delimited list:

```
1 select deptno,
2        group_concat(ename order by empno separator, ',') as emps
3   from emp
4  group by deptno
```

Oracle

Use the built-in function SYS_CONNECT_BY_PATH to build the delimited list:

```
 1 select deptno,
 2        ltrim(sys_connect_by_path(ename,','),',') emps
 3   from (
 4 select deptno,
 5        ename,
 6        row_number() over
 7                 (partition by deptno order by empno) rn,
 8        count(*) over
 9                 (partition by deptno) cnt
10   from emp
11        )
12  where level = cnt
13  start with rn = 1
14 connect by prior deptno = deptno and prior rn = rn-1
```

PostgreSQL and SQL Server

```
1 select deptno,
2        string_agg(ename order by empno separator, ',') as emps
3   from emp
4  group by deptno
```

Discussion

Being able to create delimited lists in SQL is useful because it is a common requirement. The SQL:2016 standard added LIST_AGG to perform this task, but only DB2 has implemented this function so far. Thankfully, other DBMS have similar functions, often with simpler syntax.

MySQL

The function GROUP_CONCAT in MySQL concatenates the values found in the column passed to it, in this case ENAME. It's an aggregate function, thus the need for GROUP BY in the query.

PostgreSQL and SQL Server

The STRING_AGG function syntax is similar enough to GROUP_CONCAT that the same query can be used with the GROUP_CONCAT simply changed to STRING_AGG.

Oracle

The first step to understanding the Oracle query is to break it down. Running the inline view by itself (lines 4–10), you generate a result set that includes the following for each employee: her department, her name, a rank within her respective department that is derived by an ascending sort on EMPNO, and a count of all employees in her department. For example:

```
select deptno,
        ename,
        row_number() over
                  (partition by deptno order by empno) rn,
        count(*) over (partition by deptno) cnt
  from emp

DEPTNO ENAME      RN CNT
------ ---------- -- ---
    10 CLARK       1   3
    10 KING        2   3
    10 MILLER      3   3
    20 SMITH       1   5
    20 JONES       2   5
    20 SCOTT       3   5
    20 ADAMS       4   5
```

```
20 FORD        5   5
30 ALLEN       1   6
30 WARD        2   6
30 MARTIN      3   6
30 BLAKE       4   6
30 TURNER      5   6
30 JAMES       6   6
```

The purpose of the rank (aliased RN in the query) is to allow you to walk the tree. Since the function ROW_NUMBER generates an enumeration starting from one with no duplicates or gaps, just subtract one (from the current value) to reference a prior (or parent) row. For example, the number prior to 3 is 3 minus 1, which equals 2. In this context, 2 is the parent of 3; you can observe this on line 12. Additionally, the lines:

```
start with rn = 1
connect by prior deptno = deptno
```

identify the root for each DEPTNO as having RN equal to 1 and create a new list whenever a new department is encountered (whenever a new occurrence of 1 is found for RN).

At this point, it's important to stop and look at the ORDER BY portion of the ROW_NUMBER function. Keep in mind the names are ranked by EMPNO, and the list will be created in that order. The number of employees per department is calculated (aliased CNT) and is used to ensure that the query returns only the list that has all the employee names for a department. This is done because SYS_CONNECT_BY_PATH builds the list iteratively, and you do not want to end up with partial lists.

For hierarchical queries, the pseudocolumn LEVEL starts with 1 (for queries not using CONNECT BY, LEVEL is 0, unless you are on release 10g and later when LEVEL is available only when using CONNECT BY) and increments by one after each employee in a department has been evaluated (for each level of depth in the hierarchy). Because of this, you know that once LEVEL reaches CNT, you have reached the last EMPNO and will have a complete list.

The SYS_CONNECT_BY_PATH function prefixes the list with your chosen delimiter (in this case, a comma). You may or may not want that behavior. In this recipe's solution, the call to the function LTRIM removes the leading comma from the list.

6.11 Converting Delimited Data into a Multivalued IN-List

Problem

You have delimited data that you want to pass to the IN-list iterator of a WHERE clause. Consider the following string:

```
7654,7698,7782,7788
```

You would like to use the string in a WHERE clause, but the following SQL fails because EMPNO is a numeric column:

```
select ename,sal,deptno
  from emp
 where empno in ( '7654,7698,7782,7788' )
```

This SQL fails because, while EMPNO is a numeric column, the IN list is composed of a single string value. You want that string to be treated as a comma-delimited list of numeric values.

Solution

On the surface it may seem that SQL should do the work of treating a delimited string as a list of delimited values for you, but that is not the case. When a comma embedded within quotes is encountered, SQL can't possibly know that signals a multivalued list. SQL must treat everything between the quotes as a single entity, as one string value. You must break the string up into individual EMPNOs. The key to this solution is to walk the string, but not into individual characters. You want to walk the string into valid EMPNO values.

DB2

By walking the string passed to the IN-list, you can easily convert it to rows. The functions ROW_NUMBER, LOCATE, and SUBSTR are particularly useful here:

```
 1 select empno,ename,sal,deptno
 2   from emp
 3  where empno in (
 4 select cast(substr(c,2,locate(',',c,2)-2) as integer) empno
 5   from (
 6 select substr(csv.emps,cast(iter.pos as integer)) as c
 7   from (select ','||'7654,7698,7782,7788'||',' emps
 8           from t1) csv,
 9        (select id as pos
10           from t100 ) iter
11  where iter.pos <= length(csv.emps)
12        ) x
13  where length(c) > 1
14    and substr(c,1,1) = ','
15        )
```

MySQL

By walking the string passed to the IN-list, you can easily convert it to rows:

```
select empno, ename, sal, deptno
  from emp
 where empno in
       (
select substring_index(
       substring_index(list.vals,',',iter.pos),',',-1) empno
  from (select id pos from t10) as iter,
       (select '7654,7698,7782,7788' as vals
          from t1) list
  where iter.pos <=
       (length(list.vals)-length(replace(list.vals,',','')))+1
       )
```

Oracle

By walking the string passed to the IN-list, you can easily convert it to rows. The functions ROWNUM, SUBSTR, and INSTR are particularly useful here:

```
select empno,ename,sal,deptno
  from emp
 where empno in (
       select to_number(
                 rtrim(
                substr(emps,
                 instr(emps,',',1,iter.pos)+1,
                 instr(emps,',',1,iter.pos+1)
                 instr(emps,',',1,iter.pos)),',')) emps
         from (select ','||'7654,7698,7782,7788'||',' emps from t1) csv,
              (select rownum pos from emp) iter
        where iter.pos <= ((length(csv.emps)-
                 length(replace(csv.emps,',')))/length(','))-1
)
```

PostgreSQL

By walking the string passed to the IN-list, you can easily convert it to rows. The function SPLIT_PART makes it easy to parse the string into individual numbers:

```
select ename,sal,deptno
  from emp
 where empno in (
select cast(empno as integer) as empno
  from (
select split_part(list.vals,',',iter.pos) as empno
  from (select id as pos from t10) iter,
       (select ','||'7654,7698,7782,7788'||',' as vals
          from t1) list
 where iter.pos <=
       length(list.vals)-length(replace(list.vals,',',''))
```

```
12        ) z
13  where length(empno) > 0
14        )
```

SQL Server

By walking the string passed to the IN-list, you can easily convert it to rows. The functions ROW_NUMBER, CHARINDEX, and SUBSTRING are particularly useful here:

```
 1 select empno,ename,sal,deptno
 2   from emp
 3  where empno in (select substring(c,2,charindex(',',c,2)-2) as empno
 4   from (
 5 select substring(csv.emps,iter.pos,len(csv.emps)) as c
 6   from (select ','+'7654,7698,7782,7788'+',' as emps
 7           from t1) csv,
 8        (select id as pos
 9          from t100) iter
10  where iter.pos <= len(csv.emps)
11       ) x
12  where len(c) > 1
13    and substring(c,1,1) = ','
14       )
```

Discussion

The first and most important step in this solution is to walk the string. Once you've accomplished that, all that's left is to parse the string into individual numeric values using your DBMS's provided functions.

DB2 and SQL Server

The inline view X (lines 6–11) walks the string. The idea in this solution is to "walk through" the string so that each row has one less character than the one before it:

```
,7654,7698,7782,7788,
7654,7698,7782,7788,
654,7698,7782,7788,
54,7698,7782,7788,
4,7698,7782,7788,
,7698,7782,7788,
7698,7782,7788,
698,7782,7788,
98,7782,7788,
8,7782,7788,
,7782,7788,
7782,7788,
782,7788,
82,7788,
2,7788,
```

```
,7788,
7788,
788,
88,
8,
,
```

Notice that by enclosing the string in commas (the delimiter), there's no need to make special checks as to where the beginning or end of the string is.

The next step is to keep only the values you want to use in the IN-list. The values to keep are the ones with leading commas, with the exception of the last row with its lone comma. Use SUBSTR or SUBSTRING to identify which rows have a leading comma, then keep all characters found before the next comma in that row. Once that's done, cast the string to a number so it can be properly evaluated against the numeric column EMPNO (lines 4–14):

```
 EMPNO
------
  7654
  7698
  7782
  7788
```

The final step is to use the results in a subquery to return the desired rows.

MySQL

The inline view (lines 5–9) walks the string. The expression on line 10 determines how many values are in the string by finding the number of commas (the delimiter) and adding one. The function SUBSTRING_INDEX (line 6) returns all characters in the string before (to the left of) the *n*th occurrence of a comma (the delimiter):

```
+---------------------+
| empno               |
+---------------------+
| 7654                |
| 7654,7698           |
| 7654,7698,7782      |
| 7654,7698,7782,7788 |
+---------------------+
```

Those rows are then passed to another call to SUBSTRING_INDEX (line 5); this time the *n*th occurrence of the delimited is –1, which causes all values to the right of the *n*th occurrence of the delimiter to be kept:

```
+-------+
| empno |
+-------+
| 7654  |
| 7698  |
| 7782  |
| 7788  |
+-------+
```

The final step is to plug the results into a subquery.

Oracle

The first step is to walk the string:

```
select emps,pos
  from (select ','||'7654,7698,7782,7788'||',' emps
          from t1) csv,
       (select rownum pos from emp) iter
 where iter.pos <=
 ((length(csv.emps)-length(replace(csv.emps,',')))/length(','))-1


EMPS                        POS
-------------------- ----------
,7654,7698,7782,7788,          1
,7654,7698,7782,7788,          2
,7654,7698,7782,7788,          3
,7654,7698,7782,7788,          4
```

The number of rows returned represents the number of values in your list. The values for POS are crucial to the query as they are needed to parse the string into individual values. The strings are parsed using SUBSTR and INSTR. POS is used to locate the *n*th occurrence of the delimiter in each string. By enclosing the strings in commas, no special checks are necessary to determine the beginning or end of a string. The values passed to SUBSTR and INSTR (lines 7–9) locate the *n*th and *n*th+1 occurrence of the delimiter. By subtracting the value returned for the current comma (the location in the string where the current comma is) from the value returned by the next comma (the location in the string where the next comma is) you can extract each value from the string:

```
select substr(emps,
       instr(emps,',',1,iter.pos)+1,
       instr(emps,',',1,iter.pos+1)
       instr(emps,',',1,iter.pos)) emps
  from (select ','||'7654,7698,7782,7788'||',' emps
          from t1) csv,
       (select rownum pos from emp) iter
 where iter.pos <=
  ((length(csv.emps)-length(replace(csv.emps,',')))/length(','))-1
```

```
EMPS
-----------
7654,
7698,
7782,
7788,
```

The final step is to remove the trailing comma from each value, cast it to a number, and plug it into a subquery.

PostgreSQL

The inline view Z (lines 6–9) walks the string. The number of rows returned is determined by how many values are in the string. To find the number of values in the string, subtract the size of the string without the delimiter from the size of the string with the delimiter (line 9). The function SPLIT_PART does the work of parsing the string. It looks for the value that comes before the *n*th occurrence of the delimiter:

```
select list.vals,
       split_part(list.vals,',',iter.pos) as empno,
       iter.pos
  from (select id as pos from t10) iter,
       (select ','||'7654,7698,7782,7788'||',' as vals
          from t1) list
 where iter.pos <=
       length(list.vals)-length(replace(list.vals,',',''))

     vals             | empno | pos
----------------------+-------+-----
,7654,7698,7782,7788, |       |   1
,7654,7698,7782,7788, | 7654  |   2
,7654,7698,7782,7788, | 7698  |   3
,7654,7698,7782,7788, | 7782  |   4
,7654,7698,7782,7788, | 7788  |   5
```

The final step is to cast the values (EMPNO) to a number and plug it into a subquery.

6.12 Alphabetizing a String

Problem

You want alphabetize the individual characters within strings in your tables. Consider the following result set:

```
ENAME
----------
ADAMS
ALLEN
BLAKE
CLARK
```

```
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD
```

You would like the result to be:

```
OLD_NAME   NEW_NAME
---------- --------
ADAMS      AADMS
ALLEN      AELLN
BLAKE      ABEKL
CLARK      ACKLR
FORD       DFOR
JAMES      AEJMS
JONES      EJNOS
KING       GIKN
MARTIN     AIMNRT
MILLER     EILLMR
SCOTT      COSTT
SMITH      HIMST
TURNER     ENRRTU
WARD       ADRW
```

Solution

This problem is a good example of the way increased standardization allows for more similar, and therefore portable solutions.

DB2

To alphabetize rows of strings, it is necessary to walk each string and then order its characters:

```
select ename,
       listagg(c,'')  WITHIN GROUP( ORDER BY c)
  from (
        select a.ename,
        substr(a.ename,iter.pos,1
        ) as c
 from emp a,
     (select id as pos from t10) iter
       where iter.pos <= length(a.ename)
        order by 1,2
      ) x
       Group By c
```

MySQL

The key here is the GROUP_CONCAT function, which allows you to not only concatenate the characters that make up each name but also order them:

```
1 select ename, group_concat(c order by c separator '')
2   from (
3 select ename, substr(a.ename,iter.pos,1) c
4   from emp a,
5        ( select id pos from t10 ) iter
6  where iter.pos <= length(a.ename)
7        ) x
8  group by ename
```

Oracle

The function SYS_CONNECT_BY_PATH allows you to iteratively build a list:

```
 1 select old_name, new_name
 2   from (
 3 select old_name, replace(sys_connect_by_path(c,' '),' ') new_name
 4   from (
 5 select e.ename old_name,
 6        row_number() over(partition by e.ename
 7                          order by substr(e.ename,iter.pos,1)) rn,
 8        substr(e.ename,iter.pos,1) c
 9   from emp e,
10        ( select rownum pos from emp ) iter
11  where iter.pos <= length(e.ename)
12  order by 1
13         ) x
14  start with rn = 1
15 connect by prior rn = rn-1 and prior old_name = old_name
16         )
17  where length(old_name) = length(new_name)
```

PostgreSQL

PostgreSQL has now added STRING_AGG to order characters within a string.

```
          select ename, string_agg(c , ''
                                   ORDER BY c)
from (
          select a.ename,
                 substr(a.ename,iter.pos,1) as c
            from emp a,
                 (select id as pos from t10) iter
           where iter.pos <= length(a.ename)
           order by 1,2
                 ) x
                 Group By c
```

SQL Server

If you are using SQL Server 2017 or beyond, the PostgreSQL solution with STRING_AGG will work. Otherwise, to alphabetize rows of strings, it is necessary to walk each string and then order their characters:

```
 1 select ename,
 2          max(case when pos=1 then c else '' end)+
 3          max(case when pos=2 then c else '' end)+
 4          max(case when pos=3 then c else '' end)+
 5          max(case when pos=4 then c else '' end)+
 6          max(case when pos=5 then c else '' end)+
 7          max(case when pos=6 then c else '' end)
 8     from (
 9   select e.ename,
10        substring(e.ename,iter.pos,1) as c,
11        row_number() over (
12         partition by e.ename
13             order by substring(e.ename,iter.pos,1)) as pos
14   from emp e,
15        (select row_number()over(order by ename) as pos
16           from emp) iter
17  where iter.pos <= len(e.ename)
18         ) x
19  group by ename
```

Discussion

SQL Server

The inline view X returns each character in each name as a row. The function SUBSTR or SUBSTRING extracts each character from each name, and the function ROW_NUMBER ranks each character alphabetically:

```
ENAME  C  POS
-----  -  ---
ADAMS  A  1
ADAMS  A  2
ADAMS  D  3
ADAMS  M  4
ADAMS  S  5
…
```

To return each letter of a string as a row, you must walk the string. This is accomplished with inline view ITER.

Now that the letters in each name have been alphabetized, the last step is to put those letters back together, into a string, in the order they are ranked. Each letter's position is evaluated by the CASE statements (lines 2–7). If a character is found at a particular position, it is then concatenated to the result of the next evaluation (the following CASE statement). Because the aggregate function MAX is used as well, only one

character per position POS is returned so that only one row per name is returned. The CASE evaluation goes up to the number six, which is the maximum number of characters in any name in table EMP.

MySQL

The inline view X (lines 3–6) returns each character in each name as a row. The function SUBSTR extracts each character from each name:

```
ENAME  C
-----  -
ADAMS  A
ADAMS  A
ADAMS  D
ADAMS  M
ADAMS  S
…
```

Inline view ITER is used to walk the string. From there, the rest of the work is done by the GROUP_CONCAT function. By specifying an order, the function not only concatenates each letter, it does so alphabetically.

Oracle

The real work is done by inline view X (lines 5–11), where the characters in each name are extracted and put into alphabetical order. This is accomplished by walking the string and then imposing order on those characters. The rest of the query merely glues the names back together.

The tearing apart of names can be seen by executing only inline view X:

```
OLD_NAME          RN C
---------- --------- -
ADAMS              1 A
ADAMS              2 A
ADAMS              3 D
ADAMS              4 M
ADAMS              5 S
…
```

The next step is to take the alphabetized characters and rebuild each name. This is done with the function SYS_CONNECT_BY_PATH by appending each character to the ones before it:

```
OLD_NAME   NEW_NAME
---------- ---------
ADAMS      A
ADAMS      AA
ADAMS      AAD
ADAMS      AADM
ADAMS      AADMS
…
```

The final step is to keep only the strings that have the same length as the names they were built from.

PostgreSQL

For readability, view V is used in this solution to walk the string. The function SUBSTR, in the view definition, extracts each character from each name so that the view returns:

```
ENAME C
----- -
ADAMS A
ADAMS A
ADAMS D
ADAMS M
ADAMS S
…
```

The view also orders the results by ENAME and by each letter in each name. The inline view X (lines 15–18) returns the names and characters from view V, the number of times each character occurs in each name, and its position (alphabetically):

```
ename | c | cnt | pos
------+---+-----+-----
ADAMS | A |   2 |   1
ADAMS | A |   2 |   1
ADAMS | D |   1 |   3
ADAMS | M |   1 |   4
ADAMS | S |   1 |   5
```

The extra columns CNT and POS, returned by the inline view X, are crucial to the solution. POS is used to rank each character, and CNT is used to determine the number of times the character exists in each name. The final step is to evaluate the position of each character and rebuild the name. You'll notice that each case statement is actually two case statements. This is to determine whether a character occurs more than once in a name; if it does, then rather than return that character, what is returned is that character appended to itself CNT times. The aggregate function, MAX, is used to ensure there is only one row per name.

6.13 Identifying Strings That Can Be Treated as Numbers

Problem

You have a column that is defined to hold character data. Unfortunately, the rows contain mixed numeric and character data. Consider view V:

```
create view V as
select replace(mixed,' ','') as mixed
  from (
select substr(ename,1,2)||
       cast(deptno as char(4))||
       substr(ename,3,2) as mixed
  from emp
 where deptno = 10
 union all
select cast(empno as char(4)) as mixed
  from emp
 where deptno = 20
 union all
select ename as mixed
  from emp
 where deptno = 30
       ) x
select * from v

 MIXED
 --------------
 CL10AR
 KI10NG
 MI10LL
 7369
 7566
 7788
 7876
 7902
 ALLEN
 WARD
 MARTIN
 BLAKE
 TURNER
 JAMES
```

You want to return rows that are numbers only, or that contain at least one number. If the numbers are mixed with character data, you want to remove the characters and return only the numbers. For the sample data shown previously, you want the following result set:

```
   MIXED
--------
      10
      10
      10
    7369
    7566
    7788
    7876
    7902
```

Solution

The functions REPLACE and TRANSLATE are extremely useful for manipulating strings and individual characters. The key is to convert all numbers to a single character, which then makes it easy to isolate and identify any number by referring to a single character.

DB2

Use functions TRANSLATE, REPLACE, and POSSTR to isolate the numeric characters in each row. The calls to CAST are necessary in view V; otherwise, the view will fail to be created due to type conversion errors. You'll need the function REPLACE to remove extraneous whitespace due to casting to the fixed-length CHAR:

```
 1 select mixed old,
 2        cast(
 3          case
 4          when
 5            replace(
 6           translate(mixed,'9999999999','0123456789'),'9','') = ''
 7          then
 8             mixed
 9          else replace(
10            translate(mixed,
11               repeat('#',length(mixed)),
12             replace(
13              translate(mixed,'9999999999','0123456789'),'9','')),
14                      '#','')
15          end as integer ) mixed
16   from V
17  where posstr(translate(mixed,'9999999999','0123456789'),'9') > 0
```

MySQL

The syntax for MySQL is slightly different and will define view V as:

```
create view V as
select concat(
         substr(ename,1,2),
         replace(cast(deptno as char(4)),' ',''),
```

```
          substr(ename,3,2)
        ) as mixed
  from emp
 where deptno = 10
 union all
select replace(cast(empno as char(4)), ' ', '')
  from emp where deptno = 20
 union all
select ename from emp where deptno = 30
```

Because MySQL does not support the TRANSLATE function, you must walk each row and evaluate it on a character-by-character basis.

```
 1 select cast(group_concat(c order by pos separator '') as unsigned)
 2        as MIXED1
 3   from (
 4 select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
 5   from V,
 6        ( select id pos from t10 ) iter
 7  where iter.pos <= length(v.mixed)
 8    and ascii(substr(v.mixed,iter.pos,1)) between 48 and 57
 9        ) y
10  group by mixed
11  order by 1
```

Oracle

Use functions TRANSLATE, REPLACE, and INSTR to isolate the numeric characters in each row. The calls to CAST are not necessary in view V. Use the function REPLACE to remove extraneous whitespace due to casting to the fixed-length CHAR. If you decide you would like to keep the explicit type conversion calls in the view definition, it is suggested you cast to VARCHAR2:

```
 1 select to_number (
 2         case
 3         when
 4            replace(translate(mixed,'0123456789','9999999999'),'9')
 5           is not null
 6         then
 7              replace(
 8            translate(mixed,
 9              replace(
10            translate(mixed,'0123456789','9999999999'),'9'),
11                     rpad('#',length(mixed),'#')),'#')
12         else
13              mixed
14         end
15         ) mixed
16   from V
17  where instr(translate(mixed,'0123456789','9999999999'),'9') > 0
```

PostgreSQL

Use functions TRANSLATE, REPLACE, and STRPOS to isolate the numeric characters in each row. The calls to CAST are not necessary in view V. Use the function REPLACE to remove extraneous whitespace due to casting to the fixed-length CHAR. If you decide you would like to keep the explicit type conversion calls in the view definition, it is suggested you cast to VARCHAR:

```
 1 select cast(
 2        case
 3        when
 4         replace(translate(mixed,'0123456789','9999999999'),'9','')
 5         is not null
 6        then
 7           replace(
 8        translate(mixed,
 9           replace(
10        translate(mixed,'0123456789','9999999999'),'9',''),
11                 rpad('#',length(mixed),'#')),'#','')
12        else
13          mixed
14        end as integer ) as mixed
15    from V
16  where strpos(translate(mixed,'0123456789','9999999999'),'9') > 0
```

SQL Server

The built-in function ISNUMERIC along with a wildcard search allows you to easily identify strings that contain numbers, but getting numeric characters out of a string is not particularly efficient because the TRANSLATE function is not supported.

Discussion

The TRANSLATE function is useful here as it allows you to easily isolate and identify numbers and characters. The trick is to convert all numbers to a single character; this way, rather than searching for different numbers, you search for only one character.

DB2, Oracle, and PostgreSQL

The syntax differs slightly among these DBMSs, but the technique is the same. We'll use the solution for PostgreSQL for the discussion.

The real work is done by functions TRANSLATE and REPLACE. Getting the final result set requires several function calls, each listed here in one query:

```
select mixed as orig,
translate(mixed,'0123456789','9999999999') as mixed1,
replace(translate(mixed,'0123456789','9999999999'),'9','') as mixed2,
 translate(mixed,
 replace(
```

```
 translate(mixed,'0123456789','9999999999'),'9',''),
         rpad('#',length(mixed),'#')) as mixed3,
 replace(
 translate(mixed,
 replace(
translate(mixed,'0123456789','9999999999'),'9',''),
         rpad('#',length(mixed),'#')),'#','') as mixed4
  from V
 where strpos(translate(mixed,'0123456789','9999999999'),'9') > 0

  ORIG  | MIXED1 | MIXED2 | MIXED3 | MIXED4 | MIXED5
--------+--------+--------+--------+--------+--------
 CL10AR | CL99AR | CLAR   | ##10## | 10     |     10
 KI10NG | KI99NG | KING   | ##10## | 10     |     10
 MI10LL | MI99LL | MILL   | ##10## | 10     |     10
 7369   | 9999   |        | 7369   | 7369   |   7369
 7566   | 9999   |        | 7566   | 7566   |   7566
 7788   | 9999   |        | 7788   | 7788   |   7788
 7876   | 9999   |        | 7876   | 7876   |   7876
 7902   | 9999   |        | 7902   | 7902   |   7902
```

First, notice that any rows without at least one number are removed. How this is accomplished will become clear as you examine each of the columns in the previous result set. The rows that are kept are the values in the ORIG column and are the rows that will eventually make up the result set. The first step to extracting the numbers is to use the function TRANSLATE to convert any number to a 9 (you can use any digit; 9 is arbitrary); this is represented by the values in MIXED1. Now that all numbers are 9s, they can be treated as a single unit. The next step is to remove all of the numbers by using the function REPLACE. Because all digits are now 9, REPLACE simply looks for any 9s and removes them. This is represented by the values in MIXED2. The next step, MIXED3, uses values that are returned by MIXED2. These values are then compared to the values in ORIG. If any characters from MIXED2 are found in ORIG, they are converted to the # character by TRANSLATE. The result set from MIXED3 shows that the letters, not the numbers, have now been singled out and converted to a single character. Now that all nonnumeric characters are represented by #s, they can be treated as a single unit. The next step, MIXED4, uses REPLACE to find and remove any # characters in each row; what's left are numbers only. The final step is to cast the numeric characters as numbers. Now that you've gone through the steps, you can see how the WHERE clause works. The results from MIXED1 are passed to STRPOS, and if a 9 is found (the position in the string where the first 9 is located), the result must be greater than 0. For rows that return a value greater than zero, it means there's at least one number in that row and it should be kept.

MySQL

The first step is to walk each string, evaluate each character, and determine whether it's a number:

```
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
      ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
 order by 1,2

+--------+------+------+
| mixed  | pos  | c    |
+--------+------+------+
| 7369   |    1 | 7    |
| 7369   |    2 | 3    |
| 7369   |    3 | 6    |
| 7369   |    4 | 9    |
…
| ALLEN  |    1 | A    |
| ALLEN  |    2 | L    |
| ALLEN  |    3 | L    |
| ALLEN  |    4 | E    |
| ALLEN  |    5 | N    |
…
| CL10AR |    1 | C    |
| CL10AR |    2 | L    |
| CL10AR |    3 | 1    |
| CL10AR |    4 | 0    |
| CL10AR |    5 | A    |
| CL10AR |    6 | R    |
+--------+------+------+
```

Now that each character in each string can be evaluated individually, the next step is to keep only the rows that have a number in the C column:

```
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
   from V,
       ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
  and ascii(substr(v.mixed,iter.pos,1)) between 48 and 57
 order by 1,2

+--------+------+------+
| mixed  | pos  | c    |
+--------+------+------+
| 7369   |    1 | 7    |
| 7369   |    2 | 3    |
| 7369   |    3 | 6    |
| 7369   |    4 | 9    |
…
| CL10AR |    3 | 1    |
| CL10AR |    4 | 0    |
…
+--------+------+------+
```

At this point, all the rows in column C are numbers. The next step is to use GROUP_CONCAT to concatenate the numbers to form their respective whole number in MIXED. The final result is then cast as a number:

```
select cast(group_concat(c order by pos separator '') as unsigned)
         as MIXED1
  from (
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
      ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
  and ascii(substr(x.mixed,iter.pos,1)) between 48 and 57
       ) y
  group by mixed
  order by 1

+--------+
| MIXED1 |
+--------+
|    10  |
|    10  |
|    10  |
|  7369  |
|  7566  |
|  7788  |
|  7876  |
|  7902  |
+--------+
```

As a final note, keep in mind that any digits in each string will be concatenated to form one numeric value. For example, an input value of, say, `99Gennick87` will result in the value 9987 being returned. This is something to keep in mind, particularly when working with serialized data.

6.14 Extracting the nth Delimited Substring

Problem

You want to extract a specified, delimited substring from a string. Consider the following view V, which generates source data for this problem:

```
create view V as
select 'mo,larry,curly' as name
  from t1
 union all
select 'tina,gina,jaunita,regina,leena' as name
  from t1
```

Output from the view is as follows:

```
select * from v

NAME
-------------------
mo,larry,curly
tina,gina,jaunita,regina,leena
```

You would like to extract the second name in each row, so the final result set would be as follows:

```
  SUB
-----
larry
gina
```

Solution

The key to solving this problem is to return each name as an individual row while preserving the order in which the name exists in the list. Exactly how you do these things depends on which DBMS you are using.

DB2

After walking the NAMEs returned by view V, use the function ROW_NUMBER to keep only the second name from each string:

```
 1 select substr(c,2,locate(',',c,2)-2)
 2   from (
 3 select pos, name, substr(name, pos) c,
 4        row_number() over( partition by name
 5                       order by length(substr(name,pos)) desc) rn
 6  from (
 7 select ',' ||csv.name|| ',' as name,
 8         cast(iter.pos as integer) as pos
 9   from V csv,
10        (select row_number() over() pos from t100 ) iter
11  where iter.pos <= length(csv.name)+2
12        ) x
13  where length(substr(name,pos)) > 1
14   and substr(substr(name,pos),1,1) = ','
15        ) y
16  where rn = 2
```

MySQL

After walking the NAMEs returned by view V, use the position of the commas to return only the second name in each string:

```
 1 select name
 2   from (
 3 select iter.pos,
 4        substring_index(
 5        substring_index(src.name,',',iter.pos),',',-1) name
 6   from V src,
 7        (select id pos from t10) iter,
 8  where iter.pos <=
 9        length(src.name)-length(replace(src.name,',',''))
10        ) x
11  where pos = 2
```

Oracle

After walking the NAMEs returned by view V, retrieve the second name in each list by using SUBSTR and INSTR:

```
 1 select sub
 2   from (
 3 select iter.pos,
 4        src.name,
 5        substr( src.name,
 6         instr( src.name,',',1,iter.pos )+1,
 7         instr( src.name,',',1,iter.pos+1 ) -
 8         instr( src.name,',',1,iter.pos )-1) sub
 9   from (select ','||name||',' as name from V) src,
10        (select rownum pos from emp) iter
11  where iter.pos < length(src.name)-length(replace(src.name,','))
12        )
13  where pos = 2
```

PostgreSQL

Use the function SPLIT_PART to help return each individual name as a row:

```
 1 select name
 2   from (
 3 select iter.pos, split_part(src.name,',',iter.pos) as name
 4   from (select id as pos from t10) iter,
 5        (select cast(name as text) as name from v) src
 7  where iter.pos <=
 8         length(src.name)-length(replace(src.name,',',''))+1
 9        ) x
10  where pos = 2
```

SQL Server

The SQL Server STRING_SPLIT function will do the whole job, but can only take a single cell. Hence, we use a STRING_AGG within a CTE to present the data the way STRING_SPLIT requires.

```
1 with agg_tab(name)
2     as
3     (select STRING_AGG(name,',') from V)
4 select value from
5     STRING_SPLIT(
6     (select name from agg_tab),',')
```

Discussion

DB2

The syntax is slightly different between these two DBMSs, but the technique is the same. We will use the solution for DB2 for the discussion. The strings are walked and the results are represented by inline view X:

```
select ','||csv.name|| ',' as name,
       iter.pos
  from v csv,
       (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2

EMPS                            POS
------------------------------- ----
,tina,gina,jaunita,regina,leena,    1
,tina,gina,jaunita,regina,leena,    2
,tina,gina,jaunita,regina,leena,    3
…
```

The next step is to then step through each character in each string:

```
select pos, name, substr(name, pos) c,
       row_number() over(partition by name
                         order by length(substr(name, pos)) desc) rn
  from (
select ','||csv.name||',' as name,
       cast(iter.pos as integer) as pos
  from v csv,
      (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2
       ) x
 where length(substr(name,pos)) > 1

POS EMPS            C                RN
--- --------------- ---------------- --
  1 ,mo,larry,curly, ,mo,larry,curly, 1
  2 ,mo,larry,curly, mo,larry,curly,  2
  3 ,mo,larry,curly, o,larry,curly,   3
  4 ,mo,larry,curly, ,larry,curly,    4
  …
```

Now that different portions of the string are available to you, simply identify which rows to keep. The rows you are interested in are the ones that begin with a comma; the rest can be discarded:

```
select pos, name, substr(name,pos) c,
       row_number() over(partition by name
                         order by length(substr(name, pos)) desc) rn
  from (
select ','||csv.name||',' as name,
       cast(iter.pos as integer) as pos
  from v csv,
       (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2
       ) x
 where length(substr(name,pos)) > 1
    and substr(substr(name,pos),1,1) = ','
```

```
POS  EMPS                              C                                  RN
 ---  --------------                    ----------------                    --
   1  ,mo,larry,curly,                  ,mo,larry,curly,                     1
   4  ,mo,larry,curly,                  ,larry,curly,                        2
  10  ,mo,larry,curly,                  ,curly,                              3
   1  ,tina,gina,jaunita,regina,leena,  ,tina,gina,jaunita,regina,leena,     1
   6  ,tina,gina,jaunita,regina,leena,  ,gina,jaunita,regina,leena,          2
  11  ,tina,gina,jaunita,regina,leena,  ,jaunita,regina,leena,               3
  19  ,tina,gina,jaunita,regina,leena,  ,regina,leena,                       4
  26  ,tina,gina,jaunita,regina,leena,  ,leena,                              5
```

This is an important step as it sets up how you will get the *n*th substring. Notice that many rows have been eliminated from this query because of the following condition in the WHERE clause:

```
substr(substr(name,pos),1,1) = ','
```

You'll notice that `,mo,larry,curly,` was ranked 4, but now is ranked 2. Remember, the WHERE clause is evaluated before the SELECT, so the rows with leading commas are kept, *then* ROW_NUMBER performs its ranking. At this point it's easy to see that, to get the *n*th substring, you want rows where RN equals *n*. The last step is to keep only the rows you are interested in (in this case where RN equals two) and use SUBSTR to extract the name from that row. The name to keep is the first name in the row: `larry` from `,larry,curly,` and `gina` from `,gina,jaunita,regina,leena,`.

MySQL

The inline view X walks each string. You can determine how many values are in each string by counting the delimiters in the string:

```
select iter.pos, src.name
  from (select id pos from t10) iter,
       V src
```

```
 where iter.pos <=
       length(src.name)-length(replace(src.name,',',''))

+------+-------------------------------+
| pos  | name                          |
+------+-------------------------------+
|    1 | mo,larry,curly                |
|    2 | mo,larry,curly                |
|    1 | tina,gina,jaunita,regina,leena |
|    2 | tina,gina,jaunita,regina,leena |
|    3 | tina,gina,jaunita,regina,leena |
|    4 | tina,gina,jaunita,regina,leena |
+------+-------------------------------+
```

In this case, there is one fewer row than values in each string because that's all that is needed. The function SUBSTRING_INDEX takes care of parsing the needed values:

```
select iter.pos,src.name name1,
       substring_index(src.name,',',iter.pos) name2,
       substring_index(
        substring_index(src.name,',',iter.pos),',',-1) name3
  from (select id pos from t10) iter,
       V src
 where  iter.pos <=
       length(src.name)-length(replace(src.name,',',''))

+------+-------------------------------+-------------------------+---------+
| pos  | name1                         | name2                   | name3   |
+------+-------------------------------+-------------------------+---------+
|    1 | mo,larry,curly                | mo                      | mo      |
|    2 | mo,larry,curly                | mo,larry                | larry   |
|    1 | tina,gina,jaunita,regina,leena | tina                    | tina    |
|    2 | tina,gina,jaunita,regina,leena | tina,gina               | gina    |
|    3 | tina,gina,jaunita,regina,leena | tina,gina,jaunita       | jaunita |
|    4 | tina,gina,jaunita,regina,leena | tina,gina,jaunita,regina | regina  |
+------+-------------------------------+-------------------------+---------+
```

We've shown three name fields, so you can see how the nested SUBSTRING_INDEX calls work. The inner call returns all characters to the left of the *n*th occurrence of a comma. The outer call returns everything to the right of the first comma it finds (starting from the end of the string). The final step is to keep the value for NAME3 where POS equals *n*, in this case 2.

SQL Server

STRING_SPLIT is the workhorse here, but needs its data the right way. The CTE is merely to turn the two rows of the V.names column into a single value, as required by STRING_SPLIT being a table-valued function.

Oracle

The inline view walks each string. The number of times each string is returned is determined by how many values are in each string. The solution finds the number of values in each string by counting the number of delimiters in it. Because each string is enclosed in commas, the number of values in a string is the number of commas minus one. The strings are then UNIONed and joined to a table with a cardinality that is at least the number of values in the largest string. The functions SUBSTR and INSTR use the value of POS to parse each string:

```
select iter.pos, src.name,
       substr( src.name,
        instr( src.name,',',1,iter.pos )+1,
        instr( src.name,',',1,iter.pos+1 )
        instr( src.name,',',1,iter.pos )-1) sub
  from (select ','||name||',' as name from v) src,
       (select rownum pos from emp) iter
 where iter.pos < length(src.name)-length(replace(src.name,','))
```

```
POS NAME                            SUB
--- ------------------------------- -------------
  1 ,mo,larry,curly,                mo
  1 , tina,gina,jaunita,regina,leena, tina
  2 ,mo,larry,curly,                larry
  2 , tina,gina,jaunita,regina,leena, gina
  3 ,mo,larry,curly,                curly
  3 , tina,gina,jaunita,regina,leena, jaunita
  4 , tina,gina,jaunita,regina,leena, regina
  5 , tina,gina,jaunita,regina,leena, leena
```

The first call to INSTR within SUBSTR determines the start position of the substring to extract. The next call to INSTR within SUBSTR finds the position of the *n*th comma (same as the start position) as well the position of the *n*th + 1 comma. Subtracting the two values returns the length of the substring to extract. Because every value is parsed into its own row, simply specify WHERE POS = *n* to keep the *n*th substring (in this case, where POS = 2, so the second substring in the list).

PostgreSQL

The inline view X walks each string. The number of rows returned is determined by how many values are in each string. To find the number of values in each string, find the number of delimiters in each string and add one. The function SPLIT_PART uses the values in POS to find the *n*th occurrence of the delimiter and parse the string into values:

```
select iter.pos, src.name as name1,
        split_part(src.name,',',iter.pos) as name2
   from (select id as pos from t10) iter,
        (select cast(name as text) as name from v) src
```

```
 where iter.pos <=
       length(src.name)-length(replace(src.name,',',''))+1

 pos |             name1              |  name2
-----+--------------------------------+---------
   1 | mo,larry,curly                 | mo
   2 | mo,larry,curly                 | larry
   3 | mo,larry,curly                 | curly
   1 | tina,gina,jaunita,regina,leena | tina
   2 | tina,gina,jaunita,regina,leena | gina
   3 | tina,gina,jaunita,regina,leena | jaunita
   4 | tina,gina,jaunita,regina,leena | regina
   5 | tina,gina,jaunita,regina,leena | leena
```

We've shown NAME twice so you can see how SPLIT_PART parses each string using POS. Once each string is parsed, the final step is to keep the rows where POS equals the *n*th substring you are interested in, in this case, 2.

6.15 Parsing an IP Address

Problem

You want to parse an IP address's fields into columns. Consider the following IP address:

```
111.22.3.4
```

You would like the result of your query to be:

```
A     B    C      D
----- ----- ----- ---
111   22    3     4
```

Solution

The solution depends on the built-in functions provided by your DBMS. Regardless of your DBMS, being able to locate periods and the numbers immediately surrounding them are the keys to the solution.

DB2

Use the recursive WITH clause to simulate an iteration through the IP address while using SUBSTR to easily parse it. A leading period is added to the IP address so that every set of numbers has a period in front of it and can be treated the same way.

```
1 with x (pos,ip) as (
2   values (1,'.92.111.0.222')
3   union all
4  select pos+1,ip from x where pos+1 <= 20
5 )
```

```
 6 select max(case when rn=1 then e end) a,
 7        max(case when rn=2 then e end) b,
 8        max(case when rn=3 then e end) c,
 9        max(case when rn=4 then e end) d
10   from (
11 select pos,c,d,
12        case when posstr(d,'.') > 0 then substr(d,1,posstr(d,'.')-1)
13             else d
14        end as e,
15        row_number() over( order by pos desc) rn
16   from (
17 select pos, ip,right(ip,pos) as c, substr(right(ip,pos),2) as d
18   from x
19  where pos <= length(ip)
20   and substr(right(ip,pos),1,1) = '.'
21      ) x
22      ) y
```

MySQL

The function SUBSTR_INDEX makes parsing an IP address an easy operation:

```
1 select substring_index(substring_index(y.ip,'.',1),'.',-1) a,
2        substring_index(substring_index(y.ip,'.',2),'.',-1) b,
3        substring_index(substring_index(y.ip,'.',3),'.',-1) c,
4        substring_index(substring_index(y.ip,'.',4),'.',-1) d
5   from (select '92.111.0.2' as ip from t1) y
```

Oracle

Use the built-in function SUBSTR and INSTR to parse and navigate through the IP address:

```
1 select ip,
2        substr(ip, 1, instr(ip,'.')-1 ) a,
3        substr(ip, instr(ip,'.')+1,
4                   instr(ip,'.',1,2)-instr(ip,'.')-1 ) b,
5        substr(ip, instr(ip,'.',1,2)+1,
6                   instr(ip,'.',1,3)-instr(ip,'.',1,2)-1 ) c,
7        substr(ip, instr(ip,'.',1,3)+1 ) d
8   from (select '92.111.0.2' as ip from t1)
```

PostgreSQL

Use the built-in function SPLIT_PART to parse an IP address:

```
1 select split_part(y.ip,'.',1) as a,
2        split_part(y.ip,'.',2) as b,
3        split_part(y.ip,'.',3) as c,
4        split_part(y.ip,'.',4) as d
5   from (select cast('92.111.0.2' as text) as ip from t1) as y
```

SQL Server

Use the recursive WITH clause to simulate an iteration through the IP address while using SUBSTR to easily parse it. A leading period is added to the IP address so that every set of numbers has a period in front of it and can be treated the same way:

```
with x (pos,ip) as (
  select 1 as pos,'.92.111.0.222' as ip from t1
  union all
 select pos+1,ip from x where pos+1 <= 20
)
select max(case when rn=1 then e end) a,
       max(case when rn=2 then e end) b,
       max(case when rn=3 then e end) c,
       max(case when rn=4 then e end) d
  from  (
select pos,c,d,
       case when charindex('.',d) > 0
          then substring(d,1,charindex('.',d)-1)
          else d
      end as e,
      row_number() over(order by pos desc) rn
 from (
select pos, ip,right(ip,pos) as c,
       substring(right(ip,pos),2,len(ip)) as d
 from x
where pos <= len(ip)
  and substring(right(ip,pos),1,1) = '.'
     ) x
     ) y
```

Discussion

By using the built-in functions for your database, you can easily walk through parts of a string. The key is being able to locate each of the periods in the address. Then you can parse the numbers between each.

In Recipe 6.17 we will see how regular expressions can be used with most RDBMSs—parsing an IP address is also a good area to apply this idea.

6.16 Comparing Strings by Sound

Problem

Between spelling mistakes and legitimate ways to spell words differently, such as British versus American spelling, there are many times that two words that you want to match are represented by different strings of characters. Fortunately, SQL provides a way to represent the way words sound, which allows you to find strings that sound the same even though the underlying characters aren't identical.

For example, you have a list of authors' names, including some from an earlier era when spelling wasn't as fixed as it is now, combined with some extra misspellings and typos. The following column of names is an example:

```
 a_name
----
1 Johnson
2 Jonson
3 Jonsen
4 Jensen
5 Johnsen
6 Shakespeare
7 Shakspear
8 Shaekspir
9 Shakespar
```

Although this is likely part of a longer list, you'd like to identify which of these names are plausible phonetic matches for other names on the list. While this is an exercise where there is more than one possible solution, your solution will look something like this (the meaning of the last column will become clearer by the end of the recipe):

```
a_name1        a_name2       soundex_name
----           ----          ----
Jensen         Johnson       J525
Jensen         Jonson        J525
Jensen         Jonsen        J525
Jensen         Johnsen       J525
Johnsen        Johnson       J525
Johnsen        Jonson        J525
Johnsen        Jonsen        J525
Johnsen        Jensen        J525
...
Jonson         Jensen        J525
Jonson         Johnsen       J525
Shaekspir          Shakspear     S216
Shakespar          Shakespeare   S221
Shakespeare        Shakespar     S221
Shakspear          Shaekspir     S216
```

Solution

Use the SOUNDEX function to convert strings of characters into the way they sound when spoken in English. A simple self-join allows you to compare values from the same column.

```
1  select an1.a_name as name1, an2.a_name as name2,
2  SOUNDEX(an1.a_name) as Soundex_Name
3  from author_names an1
4  join author_names an2
5  on (SOUNDEX(an1.a_name)=SOUNDEX(an2.a_name)
6  and an1.a_name not like an2.a_name)
```

Discussion

The thinking behind SOUNDEX predates both databases and computing, as it originated with the US Census as an attempt to resolve different spellings of proper names for both people and places. There are many algorithms that attempt the same task as SOUNDEX, and, of course, there are alternative versions for languages other than English. However, we cover SOUNDEX, as it comes with most RDBMSs.

Soundex keeps the first letter of the name and then replaces the remaining values with numbers that have the same value if they are phonetically similar. For example, *m* and *n* are both replaced with the number 5.

In the previous example, the actual Soundex output is shown in the *Soundex_Name* column. This is just to show what is happening, and not necessary for the solution; some RDMSs even have a function that hides the Soundex result, such as SQL Server's `Difference` function, which compares two strings using Soundex and returns a similarity scale from 0 to 4 (e.g., 4 is a perfect match between the Soundex outputs, representing 4/4 characters in the Soundex version if the two strings match).

Sometimes Soundex will be sufficient for your needs; other times it won't be. However, a small amount of research, possibly using texts such as *Data Matching* (Christen, 2012), will help you find other algorithms that are frequently (but not always) simple to implement as a user-defined function, or in another programming language to suit your taste and needs.

6.17 Finding Text Not Matching a Pattern

Problem

You have a text field that contains some structured text values (e.g., phone numbers), and you want to find occurrences where those values are structured incorrectly. For example, you have data like the following:

```
select emp_id, text
  from employee_comment

EMP_ID     TEXT
---------- ------------------------------------------------------------
7369       126 Varnum, Edmore MI 48829, 989 313-5351
7499       1105 McConnell Court
           Cedar Lake MI 48812
           Home: 989-387-4321
           Cell: (237) 438-3333
```

and you want to list rows having invalidly formatted phone numbers. For example, you want to list the following row because its phone number uses two different separator characters:

```
7369           126 Varnum, Edmore MI 48829, 989 313-5351
```

You want to consider valid only those phone numbers that use the same character for both delimiters.

Solution

This problem has a multipart solution:

1. Find a way to describe the universe of apparent phone numbers that you want to consider.
2. Remove any validly formatted phone numbers from consideration.
3. See whether you still have any apparent phone numbers left. If you do, you know those are invalidly formatted.

```
select emp_id, text
from employee_comment
where regexp_like(text, '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
  and regexp_like(
         regexp_replace(text,
            '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}','***'),
         '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')

    EMP_ID TEXT
---------- ------------------------------------------------------------
  7369     126 Varnum, Edmore MI 48829, 989 313-5351
  7844     989-387.5359
  9999     906-387-1698, 313-535.8886
```

Each of these rows contains at least one apparent phone number that is not correctly formatted.

Discussion

The key to this solution lies in the detection of an "apparent phone number." Given that the phone numbers are stored in a comment field, any text at all in the field could be construed to be an invalid phone number. You need a way to narrow the field to a more reasonable set of values to consider. You don't, for example, want to see the following row in your output:

```
    EMP_ID TEXT
---------- ------------------------------------------------------------
      7900 Cares for 100-year-old aunt during the day. Schedule only
           for evening and night shifts.
```

Clearly there's no phone number at all in this row, much less one that is invalid. We can all see that. The question is, how do you get the RDBMS to "see" it? We think you'll enjoy the answer. Please read on.

This recipe comes (with permission) from an article by Jonathan Gennick called "Regular Expression Anti-Patterns."

The solution uses Pattern A to define the set of "apparent" phone numbers to consider:

```
Pattern A: [0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}
```

Pattern A checks for two groups of three digits followed by one group of four digits. Any one of a dash (-), a period (.), or a space is accepted as a delimiter between groups. You could come up with a more complex pattern. For example, you could decide that you also want to consider seven-digit phone numbers. But don't get sidetracked. The point now is that somehow you do need to define the universe of possible phone number strings to consider, and for this problem that universe is defined by Pattern A. You can define a different Pattern A, and the general solution still applies.

The solution uses Pattern A in the WHERE clause to ensure that only rows having potential phone numbers (as defined by the pattern!) are considered:

```
select emp_id, text
  from employee_comment
 where regexp_like(text, '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
```

Next, you need to define what a "good" phone number looks like. The solution does this using Pattern B:

```
Pattern B: [0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}
```

This time, the pattern uses \1 to reference the first subexpression. Whichever character is matched by ([-.]) must also be matched by \1. Pattern B describes good phone numbers, which must be eliminated from consideration (as they are not bad). The solution eliminates the well-formatted phone numbers through a call to REGEXP_REPLACE:

```
regexp_replace(text,
           '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}','***'),
```

This call to REGEXP_REPLACE occurs in the WHERE clause. Any well-formatted phone numbers are replaced by a string of three asterisks. Again, Pattern B can be any pattern that you desire. The point is that Pattern B describes the acceptable pattern that you are after.

Having replaced well-formatted phone numbers with strings of three asterisks (*), any "apparent" phone numbers that remain must, by definition, be poorly formatted. The solution applies REGEXP_LIKE to the output from REGEXP_LIKE to see whether any poorly formatted phone numbers remain:

```
and regexp_like(
       regexp_replace(text,
          '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}','***'),
       '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
```

Regular expressions are a big topic in their own right, requiring practice to master. Once you do master them, you will find they match a great variety of string patterns with ease. We recommend studying a book such as *Mastering Regular Expressions* by Jeffrey Friedl to get your regular expression skills to the required level.

6.18 Summing Up

Matching on strings can be a painful task. SQL has added a range of tools to reduce the pain, and mastering them will keep you out of trouble. Although a lot can be done with the native SQL string functions, using the regular expression functions that are increasingly available takes it to another level altogether.

CHAPTER 7

Working with Numbers

This chapter focuses on common operations involving numbers, including numeric computations. While SQL is not typically considered the first choice for complex computations, it is efficient for day-to-day numeric chores. More importantly, as databases and datawarehouses supporting SQL probably remain the most common place to find an organization's data, using SQL to explore and evaluate that data is essential for anyone putting that data to work. The techniques in this section have also been chosen to help data scientists decide which data is the most promising for further analysis.

Some recipes in this chapter make use of aggregate functions and the GROUP BY clause. If you are not familiar with grouping, please read at least the first major section, called "Grouping," in Appendix A.

7.1 Computing an Average

Problem

You want to compute the average value in a column, either for all rows in a table or for some subset of rows. For example, you might want to find the average salary for all employees as well as the average salary for each department.

Solution

When computing the average of all employee salaries, simply apply the AVG function to the column containing those salaries.

By excluding a WHERE clause, the average is computed against all non-NULL values:

```
1 select avg(sal) as avg_sal
2   from emp

   AVG_SAL
----------
2073.21429
```

To compute the average salary for each department, use the GROUP BY clause to create a group corresponding to each department:

```
1 select deptno, avg(sal) as avg_sal
2   from emp
3  group by deptno

    DEPTNO     AVG_SAL
----------  ----------
        10  2916.66667
        20        2175
        30  1566.66667
```

Discussion

When finding an average where the whole table is the group or window, simply apply the AVG function to the column you are interested in without using the GROUP BY clause. It is important to realize that the function AVG ignores NULLs. The effect of NULL values being ignored can be seen here:

```
create table t2(sal integer)
insert into t2 values (10)
insert into t2 values (20)
insert into t2 values (null)
select avg(sal)    select distinct 30/2
  from t2            from t2

  AVG(SAL)               30/2
----------          ----------
        15                  15


select avg(coalesce(sal,0))    select distinct 30/3
  from t2                        from t2

AVG(COALESCE(SAL,0))                 30/3
--------------------            ----------
                  10                    10
```

The COALESCE function will return the first non-NULL value found in the list of values that you pass. When NULL SAL values are converted to zero, the average changes. When invoking aggregate functions, always give thought to how you want NULLs handled.

The second part of the solution uses GROUP BY (line 3) to divide employee records into groups based on department affiliation. GROUP BY automatically causes aggregate functions such as AVG to execute and return a result for each group. In this example, AVG would execute once for each department-based group of employee records.

It is not necessary, by the way, to include GROUP BY columns in your select list. For example:

```
select avg(sal)
  from emp
 group by deptno

  AVG(SAL)
----------
2916.66667
      2175
1566.66667
```

You are still grouping by DEPTNO even though it is not in the SELECT clause. Including the column you are grouping by in the SELECT clause often improves readability, but is not mandatory. It is mandatory, however, to avoid placing columns in your SELECT list that are not also in your GROUP BY clause.

See Also

See Appendix A for a refresher on GROUP BY functionality.

7.2 Finding the Min/Max Value in a Column

Problem

You want to find the highest and lowest values in a given column. For example, you want to find the highest and lowest salaries for all employees, as well as the highest and lowest salaries for each department.

Solution

When searching for the lowest and highest salaries for all employees, simply use the functions MIN and MAX, respectively:

```
1 select min(sal) as min_sal, max(sal) as max_sal
  2   from emp

   MIN_SAL     MAX_SAL
----------  ----------
       800        5000
```

When searching for the lowest and highest salaries for each department, use the functions MIN and MAX with the GROUP BY clause:

```
1 select deptno, min(sal) as min_sal, max(sal) as max_sal
  2   from emp
  3  group by deptno

       DEPTNO     MIN_SAL     MAX_SAL
   ----------  ----------  ----------
           10        1300        5000
           20         800        3000
           30         950        2850
```

Discussion

When searching for the highest or lowest values, and in cases where the whole table is the group or window, simply apply the MIN or MAX function to the column you are interested in without using the GROUP BY clause.

Remember that the MIN and MAX functions ignore NULLs, and that you can have NULL groups as well as NULL values for columns in a group. The following are examples that ultimately lead to a query using GROUP BY that returns NULL values for two groups (DEPTNO 10 and 20):

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1


     DEPTNO       COMM
 ---------- ----------
         10
         10
         10
         30        300
         30        500
         30
         30          0
         30       1300
         30


select min(comm), max(comm)
  from emp

 MIN(COMM)   MAX(COMM)
----------  ----------
         0        1300
```

```
select deptno, min(comm), max(comm)
  from emp
 group by deptno

     DEPTNO  MIN(COMM)   MAX(COMM)
 ---------- ----------  ----------
         10
         20
         30          0        1300
```

Remember, as Appendix A points out, even if nothing other than aggregate functions are listed in the SELECT clause, you can still group by other columns in the table; for example:

```
select min(comm), max(comm)
  from emp
 group by deptno

 MIN(COMM)   MAX(COMM)
----------  ----------
         0        1300
```

Here you are still grouping by DEPTNO even though it is not in the SELECT clause. Including the column you are grouping by in the SELECT clause often improves readability, but is not mandatory. It is mandatory, however, that any column in the SELECT list of a GROUP BY query also be listed in the GROUP BY clause.

See Also

See Appendix A for a refresher on GROUP BY functionality.

7.3 Summing the Values in a Column

Problem

You want to compute the sum of all values, such as all employee salaries, in a column.

Solution

When computing a sum where the whole table is the group or window, just apply the SUM function to the columns you are interested in without using the GROUP BY clause:

```
1 select sum(sal)
2  from emp

  SUM(SAL)
----------
     29025
```

When creating multiple groups or windows of data, use the SUM function with the GROUP BY clause. The following example sums employee salaries by department:

```
1 select deptno, sum(sal) as total_for_dept
2   from emp
3  group by deptno


    DEPTNO  TOTAL_FOR_DEPT
----------  --------------
        10           8750
        20          10875
        30           9400
```

Discussion

When searching for the sum of all salaries for each department, you are creating groups or "windows" of data. Each employee's salary is added together to produce a total for their respective department. This is an example of aggregation in SQL because detailed information, such as each individual employee's salary, is not the focus; the focus is the end result for each department. It is important to note that the SUM function will ignore NULLs, but you can have NULL groups, which can be seen here. DEPTNO 10 does not have any employees who earn a commission; thus, grouping by DEPTNO 10 while attempting to SUM the values in COMM will result in a group with a NULL value returned by SUM:

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1

    DEPTNO       COMM
---------- ----------
        10
        10
        10
        30        300
        30        500
        30
        30          0
        30       1300
        30


select sum(comm)
  from emp

 SUM(COMM)
----------
      2100
```

```
select deptno, sum(comm)
  from emp
 where deptno in (10,30)
 group by deptno

    DEPTNO   SUM(COMM)
----------  ----------
        10
        30        2100
```

See Also

See Appendix A for a refresher on GROUP BY functionality.

7.4 Counting Rows in a Table

Problem

You want to count the number of rows in a table, or you want to count the number of values in a column. For example, you want to find the total number of employees as well as the number of employees in each department.

Solution

When counting rows where the whole table is the group or window, simply use the COUNT function along with the * character:

```
1 select count(*)
2  from emp

  COUNT(*)
----------
        14
```

When creating multiple groups, or windows of data, use the COUNT function with the GROUP BY clause:

```
1 select deptno, count(*)
2   from emp
3  group by deptno

    DEPTNO    COUNT(*)
----------  ----------
        10           3
        20           5
        30           6
```

Discussion

When counting the number of employees for each department, you are creating groups or "windows" of data. Each employee found increments the count by one to produce a total for their respective department. This is an example of aggregation in SQL because detailed information, such as each individual employee's salary or job, is not the focus; the focus is the end result for each department. It is important to note that the COUNT function will ignore NULLs when passed a column name as an argument, but will include NULLs when passed the * character or any constant; consider the following:

```
select deptno, comm
  from emp

    DEPTNO        COMM
----------  ----------
        20
        30         300
        30         500
        20
        30        1300
        30
        10
        20
        10
        30           0
        20
        30
        20
        10


select count(*), count(deptno), count(comm), count('hello')
  from emp

  COUNT(*)  COUNT(DEPTNO)   COUNT(COMM)  COUNT('HELLO')
----------  -------------   -----------  --------------
        14             14             4              14


select deptno, count(*), count(comm), count('hello')
  from emp
 group by deptno

     DEPTNO   COUNT(*)  COUNT(COMM)  COUNT('HELLO')
 ---------- ----------  -----------  --------------
         10          3            0               3
         20          5            0               5
         30          6            4               6
```

If all rows are null for the column passed to COUNT or if the table is empty, COUNT will return zero. It should also be noted that, even if nothing other than aggregate functions are specified in the SELECT clause, you can still group by other columns in the table, for example:

```
select count(*)
  from emp
 group by deptno

   COUNT(*)
 ----------
          3
          5
          6
```

Notice that you are still grouping by DEPTNO even though it is not in the SELECT clause. Including the column you are grouping by in the SELECT clause often improves readability, but is not mandatory. If you do include it (in the SELECT list), it is mandatory that it is listed in the GROUP BY clause.

See Also

See Appendix A for a refresher on GROUP BY functionality.

7.5 Counting Values in a Column

Problem

You want to count the number of non-NULL values in a column. For example, you'd like to find out how many employees are on commission.

Solution

Count the number of non-NULL values in the EMP table's COMM column:

```
select count(comm)
  from emp

COUNT(COMM)
-----------
          4
```

Discussion

When you "count star," as in COUNT(*), what you are really counting is rows (regardless of actual value, which is why rows containing NULL and non-NULL values are counted). But when you COUNT a column, you are counting the number of non-NULL values in that column. The previous recipe's discussion touches on this

distinction. In this solution, COUNT(COMM) returns the number of non-NULL values in the COMM column. Since only commissioned employees have commissions, the result of COUNT(COMM) is the number of such employees.

7.6 Generating a Running Total

Problem

You want to calculate a running total of values in a column.

Solution

As an example, the following solutions show how to compute a running total of salaries for all employees. For readability, results are ordered by SAL whenever possible so that you can easily eyeball the progression of the running total.

```
1 select ename, sal,
2        sum(sal) over (order by sal,empno) as running_total
3   from emp
4   order by 2


ENAME              SAL  RUNNING_TOTAL
----------  ----------  -------------
SMITH              800            800
JAMES              950           1750
ADAMS             1100           2850
WARD              1250           4100
MARTIN            1250           5350
MILLER            1300           6650
TURNER            1500           8150
ALLEN             1600           9750
CLARK             2450          12200
BLAKE             2850          15050
JONES             2975          18025
SCOTT             3000          21025
FORD              3000          24025
KING              5000          29025
```

Discussion

The windowing function SUM OVER makes generating a running total a simple task. The ORDER BY clause in the solution includes not only the SAL column, but also the EMPNO column (which is the primary key) to avoid duplicate values in the running total. The column RUNNING_TOTAL2 in the following example illustrates the problem that you might otherwise have with duplicates:

```
select empno, sal,
       sum(sal)over(order by sal,empno) as running_total1,
       sum(sal)over(order by sal) as running_total2
  from emp
 order by 2

ENAME              SAL  RUNNING_TOTAL1  RUNNING_TOTAL2
----------  ----------  --------------  --------------
SMITH              800             800             800
JAMES              950            1750            1750
ADAMS             1100            2850            2850
WARD              1250            4100            5350
MARTIN            1250            5350            5350
MILLER            1300            6650            6650
TURNER            1500            8150            8150
ALLEN             1600            9750            9750
CLARK             2450           12200           12200
BLAKE             2850           15050           15050
JONES             2975           18025           18025
SCOTT             3000           21025           24025
FORD              3000           24025           24025
KING              5000           29025           29025
```

The values in RUNNING_TOTAL2 for WARD, MARTIN, SCOTT, and FORD are incorrect. Their salaries occur more than once, and those duplicates are summed and added to the running total. This is why EMPNO (which is unique) is needed to produce the (correct) results that you see in RUNNING_TOTAL1. Consider this: for ADAMS you see 2850 for RUNNING_TOTAL1 and RUNNING_TOTAL2. Add WARD's salary of 1250 to 2850 and you get 4100, yet RUNNING_TOTAL2 returns 5350. Why? Since WARD and MARTIN have the same SAL, their two 1250 salaries are added together to yield 2500, which is then added to 2850 to arrive at 5350 for both WARD and MARTIN. By specifying a combination of columns to order by that cannot result in duplicate values (e.g., any combination of SAL and EMPNO is unique), you ensure the correct progression of the running total.

7.7 Generating a Running Product

Problem

You want to compute a running product on a numeric column. The operation is similar to Recipe 7.6, but using multiplication instead of addition.

Solution

By way of example, the solutions all compute running products of employee salaries. While a running product of salaries may not be all that useful, the technique can easily be applied to other, more useful domains.

Use the windowing function SUM OVER and take advantage of the fact that you can simulate multiplication by adding logarithms:

```
1 select empno,ename,sal,
2        exp(sum(ln(sal))over(order by sal,empno)) as running_prod
3   from emp
4  where deptno = 10

EMPNO ENAME          SAL            RUNNING_PROD
----- ----------    ----    --------------------
 7934 MILLER        1300                    1300
 7782 CLARK         2450                 3185000
 7839 KING          5000             15925000000
```

It is not valid in SQL (or, formally speaking, in mathematics) to compute logarithms of values less than or equal to zero. If you have such values in your tables, you need to avoid passing those invalid values to SQL's LN function. Precautions against invalid values and NULLs are not provided in this solution for the sake of readability, but you should consider whether to place such precautions in production code that you write. If you absolutely must work with negative and zero values, then this solution may not work for you. At the same time, if you have zeros (but no values below zero), a common workaround is to add 1 to all values, noting that the logarithm of 1 is always zero regardless of base.

SQL Server users use LOG instead of LN.

Discussion

The solution takes advantage of the fact that you can multiply two numbers by:

1. Computing their respective natural logarithms
2. Summing those logarithms
3. Raising the result to the power of the mathematical constant *e* (using the EXP function)

The one caveat when using this approach is that it doesn't work for summing zero or negative values, because any value less than or equal to zero is out of range for an SQL logarithm.

For an explanation of how the window function SUM OVER works, see Recipe 7.6.

7.8 Smoothing a Series of Values

Problem

You have a series of values that appear over time, such as monthly sales figures. As is common, the data shows a lot of variation from point to point, but you are interested in the overall trend. Therefore, you want to implement a simple smoother, such as weighted running average to better identify the trend.

Imagine you have daily sales totals, in dollars, such as from a newsstand:

```
DATE1            SALES
2020-01-01         647
2020-01-02         561
2020-01-03         741
2020-01-04         978
2020-01-05        1062
2020-01-06        1072
...                ...
```

However, you know that there is volatility to the sales data that makes it difficult to discern an underlying trend. Possibly different days of the week or month are known to have especially high or low sales. Alternatively, maybe you are aware that due to the way the data is collected, sometimes sales for one day are moved into the next day, creating a trough followed by a peak, but there is no practical way to allocate the sales to their correct day. Therefore, you need to smooth the data over a number of days to achieve a proper view of what's happening.

A moving average can be calculated by summing the current value and the preceding *n*-1 values and dividing by *n*. If you also display the previous values for reference, you expect something like this:

```
DATE1         sales    salesLagOne   SalesLagTwo    MovingAverage
-----        ------   -----------   ------------  --------------
2020-01-01    647       NULL           NULl                 NULL
2020-01-02    561       647            NULL                 NULL
2020-01-03    741       561            647               649.667
2020-01-04    978       741            561                   760
2020-01-05    1062      978            741                   927
2020-01-06    1072      1062           978              1037.333
2020-01-07    805       1072           1062              979.667
2020-01-08    662       805            1072              846.333
2020-01-09    1083      662            805                   850
2020-01-10    970       1083           662                   905
```

Solution

The formula for the mean is well known. By applying a simple weighting to the formula, we can make it more relevant for this task by giving more weight to more recent values. Use the window function LAG to create a moving average:

```
select date1, sales,lag(sales,1) over(order by date1) as salesLagOne,
lag(sales,2) over(order by date1) as salesLagTwo,
(sales
+ (lag(sales,1) over(order by date1))
+ lag(sales,2) over(order by date1))/3 as MovingAverage
from sales
```

Discussion

A weighted moving average is one of the simplest ways to analyze time-series data (data that appears at particular time intervals). This is just one way to calculate a simple moving average—you can also use a partition with average. Although we have selected a simple three-point moving average, there are different formulas with differing numbers of points according to the characteristics of the data you apply them [.keep-together]#to—#that's where this technique really comes into its own.

For example, a simple three-point weighted moving average that emphasizes the most recent data point could be implemented with the following variant on the solution, where coefficients and the denominator have been updated:

```
select date1, sales,lag(sales,1) over(order by date1),
lag(sales,2) over(order by date1),
((3*sales)
+ (2*(lag(sales,1) over(order by date1)))
+ (lag(sales,2) over(order by date1)))/6 as SalesMA
from sales
```

7.9 Calculating a Mode

Problem

You want to find the mode (for those of you who don't recall, the *mode* in mathematics is the element that appears most frequently for a given set of data) of the values in a column. For example, you want to find the mode of the salaries in DEPTNO 20.

Based on the following salaries:

```
select sal
  from emp
 where deptno = 20
 order by sal
```

```
       SAL
----------
       800
      1100
      2975
      3000
      3000
```

the mode is 3000.

Solution

DB2, MySQL, PostgreSQL, and SQL Server

Use the window function DENSE_RANK to rank the counts of the salaries to facilitate extracting the mode:

```
 1 select sal
 2   from (
 3 select sal,
 4        dense_rank()over( order by cnt desc) as rnk
 5   from (
 6 select sal, count(*) as cnt
 8   from emp
 9  where deptno = 20
10  group by sal
11        ) x
12        ) y
13  where rnk = 1
```

Oracle

You can use the KEEP extension to the aggregate function MAX to find the mode SAL. One important note is that if there are ties, i.e., multiple rows that are the mode, the solution using KEEP will keep only one, and that is the one with the highest salary. If you want to see all modes (if more than one exists), you must modify this solution or simply use the DB2 solution presented earlier. In this case, since 3000 is the mode SAL in DEPTNO 20 and is also the highest SAL, this solution is sufficient:

```
1 select max(sal)
2        keep(dense_rank first order by cnt desc) sal
3   from (
4 select sal, count(*) cnt
5   from emp
6  where deptno=20
7  group by sal
8        )
```

Discussion

DB2 and SQL Server

The inline view X returns each SAL and the number of times it occurs. Inline view Y uses the window function DENSE_RANK (which allows for ties) to sort the results.

The results are ranked based on the number of times each SAL occurs, as shown here:

```
1 select sal,
2        dense_rank()over(order by cnt desc) as rnk
3   from (
4 select sal,count(*) as cnt
5   from emp
6  where deptno = 20
7  group by sal
8        ) x

  SAL          RNK
-----   ----------
 3000            1
  800            2
 1100            2
 2975            2
```

The outermost portion of query simply keeps the row(s) where RNK is 1.

Oracle

The inline view returns each SAL and the number of times it occurs and is shown here:

```
select sal, count(*) cnt
    from emp
   where deptno=20
   group by sal

      SAL          CNT
    -----   ----------
      800            1
     1100            1
     2975            1
     3000            2
```

The next step is to use the KEEP extension of the aggregate function MAX to find the mode. If you analyze the KEEP clause shown here, you will notice three subclauses, DENSE_RANK, FIRST, and ORDER BY CNT DESC:

```
keep(dense_rank first order by cnt desc)
```

This makes finding the mode extremely convenient. The KEEP clause determines which SAL will be returned by MAX by looking at the value of CNT returned by the inline view. Working from right to left, the values for CNT are ordered in descending order; then the first is kept of all the values for CNT returned in DENSE_RANK order. Looking at the result set from the inline view, you can see that 3000 has the highest CNT of 2. The MAX(SAL) returned is the greatest SAL that has the greatest CNT, in this case 3000.

See Also

See Chapter 11, particularly the section on "Finding Knight Values," for a deeper discussion of Oracle's KEEP extension of aggregate functions.

7.10 Calculating a Median

Problem

You want to calculate the median (for those of who do not recall, the *median* is the value of the middle member of a set of ordered elements) value for a column of numeric values. For example, you want to find the median of the salaries in DEPTNO 20. Based on the following salaries:

```
select sal
  from emp
 where deptno = 20
 order by sal

       SAL
----------
       800
      1100
      2975
      3000
      3000
```

the median is 2975.

Solution

Other than the Oracle solution (which uses supplied functions to compute a median), the introduction of window functions allows for a more efficient solution compared to the traditional self-join.

DB2 and PostgreSQL

Use the window function PERCENTILE_CONT to find the median:

```
1 select percentile_cont(0.5)
2         within group(order by sal)
3   from emp
4  where deptno=20
```

SQL Server

Use the window function PERCENTILE_CONT to find the median:

```
1 select percentile_cont(0.5)
    2         within group(order by sal)
    3         over()
    4   from emp
    5  where deptno=20
```

The SQL Server solution works on the same principle but requires an OVER clause.

MySQL

MySQL doesn't have the PERCENTILE_CONT function, so a workaround is required. One way is to use the CUME_DIST function in conjunction with a CTE, effectively re-creating the PERCENTILE_CONT function:

```
with rank_tab (sal, rank_sal) as
(
select sal, cume_dist() over (order by sal)
            from emp
            where deptno=20
),

inter as
(
        select sal, rank_sal from rank_tab
        where rank_sal>=0.5
union
        select sal, rank_sal from rank_tab
        where rank_sal<=0.5
)

        select avg(sal) as MedianSal
               from inter
```

Oracle

Use the functions MEDIAN or PERCENTILE_CONT:

```
1 select median(sal)
2   from emp
3  where deptno=20

1 select percentile_cont(0.5)
2         within group(order by sal)
```

```
  3   from emp
  4  where deptno=20
```

Discussion

Oracle, PostgreSQL, SQL Server, and DB2

Other than Oracle's MEDIAN function, the structure of all the solutions is the same. The PERCENTILE_CONT function allows you to directly apply the definition of a median, as the median is by definition the 50th percentile. Hence, applying this function with the appropriate syntax and using 0.5 as the argument finds the median.

Of course, other percentiles are also available from this function. For example, you can look for the 5th and/or 95th percentiles to find outliers (another method of finding outliers is outlined later in this chapter when we discuss the median absolute deviation).

MySQL

MySQL doesn't have a PERCENTILE_CONT function, which makes things trickier. To find the median, the values for SAL must be ordered from lowest to highest. The CUME_DIST function achieves this goal and labels each row with its percentile. Hence, it can be used to achieve the same outcome as the PERCENTILE_CONT function used in the solution for the other databases.

The only difficulty is that the CUME_DIST function is not permitted in a WHERE clause. As a result, you need to apply it first in a CTE.

The only trap here is that if the number of rows is even, there won't be a row exactly on the median. Hence, the solution is written to find the average of the highest value below or equal to the median, and the lowest value above or equal to the median. This method works for both odd and even numbers of rows, and if there is an odd number of rows giving an exact median, it will take average of two numbers that are equal.

7.11 Determining the Percentage of a Total

Problem

You want to determine the percentage that values in a specific column represent against a total. For example, you want to determine what percentage of all salaries are the salaries in DEPTNO 10 (the percentage that DEPTNO 10 salaries contribute to the total).

Solution

In general, computing a percentage against a total in SQL is no different than doing so on paper: simply divide, then multiply. In this example you want to find the percentage of total salaries in table EMP that come from DEPTNO 10. To do that, simply find the salaries for DEPTNO 10, and then divide by the total salary for the table. As the last step, multiply by 100 to return a value that represents a percent.

MySQL and PostgreSQL

Divide the sum of the salaries in DEPTNO 10 by the sum of all salaries:

```
1 select (sum(
2          case when deptno = 10 then sal end)/sum(sal)
3         )*100 as pct
4   from emp
```

DB2, Oracle, and SQL Server

Use an inline view with the window function SUM OVER to find the sum of all salaries along with the sum of all salaries in DEPTNO 10. Then do the division and multiplication in the outer query:

```
1 select distinct (d10/total)*100 as pct
2   from (
3 select deptno,
4        sum(sal)over() total,
5        sum(sal)over(partition by deptno) d10
6   from emp
7        ) x
8  where deptno=10
```

Discussion

MySQL and PostgreSQL

The CASE statement conveniently returns only the salaries from DEPTNO 10. They are then summed and divided by the sum of all the salaries. Because NULLs are ignored by aggregates, an ELSE clause is not needed in the CASE statement. To see exactly which values are divided, execute the query without the division:

```
select sum(case when deptno = 10 then sal end) as d10,
       sum(sal)
  from emp

D10  SUM(SAL)
---- ---------
8750     29025
```

Depending on how you define SAL, you may need to explicitly use CAST when performing division to ensure the correct data type. For example, on DB2, SQL Server, and PostgreSQL, if SAL is stored as an integer, you can apply CAST to ensure a decimal value is returned, as shown here:

```
select (cast(
         sum(case when deptno = 10 then sal end)
            as decimal)/sum(sal)
        )*100 as pct
  from emp
```

DB2, Oracle, and SQL Server

As an alternative to the traditional solution, this solution uses window functions to compute a percentage relative to the total. For DB2 and SQL Server, if you've stored SAL as an integer, you'll need to use CAST before dividing:

```
select distinct
       cast(d10 as decimal)/total*100 as pct
  from (
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
       ) x
 where deptno=10
```

It is important to keep in mind that window functions are applied after the WHERE clause is evaluated. Thus, the filter on DEPTNO cannot be performed in inline view X. Consider the results of inline view X without and with the filter on DEPTNO. First without:

```
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp

DEPTNO      TOTAL       D10
------- --------- ---------
     10     29025      8750
     10     29025      8750
     10     29025      8750
     20     29025     10875
     20     29025     10875
     20     29025     10875
     20     29025     10875
     20     29025     10875
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
```

```
        30     29025      9400
        30     29025      9400
```

and now with:

```
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
 where deptno=10

DEPTNO     TOTAL       D10
------ --------- ---------
    10      8750      8750
    10      8750      8750
    10      8750      8750
```

Because window functions are applied after the WHERE clause, the value for TOTAL represents the sum of all salaries in DEPTNO 10 only. But to solve the problem you want the TOTAL to represent the sum of all salaries, period. That's why the filter on DEPTNO must happen outside of inline view X.

7.12 Aggregating Nullable Columns

Problem

You want to perform an aggregation on a column, but the column is nullable. You want the accuracy of your aggregation to be preserved, but are concerned because aggregate functions ignore NULLs. For example, you want to determine the average commission for employees in DEPTNO 30, but there are some employees who do not earn a commission (COMM is NULL for those employees). Because NULLs are ignored by aggregates, the accuracy of the output is compromised. You would like to somehow include NULL values in your aggregation.

Solution

Use the COALESCE function to convert NULLs to zero so they will be included in the aggregation:

```
1 select avg(coalesce(comm,0)) as avg_comm
2   from emp
3  where deptno=30
```

Discussion

When working with aggregate functions, keep in mind that NULLs are ignored. Consider the output of the solution without using the COALESCE function:

```
select avg(comm)
  from emp
 where deptno=30

 AVG(COMM)
 ---------
       550
```

This query shows an average commission of 550 for DEPTNO 30, but a quick examination of those rows:

```
select ename, comm
  from emp
 where deptno=30
order by comm desc

ENAME           COMM
---------- ---------
BLAKE
JAMES
MARTIN          1400
WARD             500
ALLEN            300
TURNER             0
```

shows that only four of the six employees can earn a commission. The sum of all commissions in DEPTNO 30 is 2200, and the average should be 2200/6, not 2200/4. By excluding the COALESCE function, you answer the question "What is the average commission of employees in DEPTNO 30 *who can earn a commission*?" rather than "What is the average commission of all employees in DEPTNO 30?" When working with aggregates, remember to treat NULLs accordingly.

7.13 Computing Averages Without High and Low Values

Problem

You want to compute an average, but you want to exclude the highest and lowest values to (hopefully) reduce the effect of skew. In statistical language, this is known as a *trimmed mean*. For example, you want to compute the average salary of all employees excluding the highest and lowest salaries.

Solution

MySQL and PostgreSQL

Use subqueries to exclude high and low values:

```
1 select avg(sal)
2   from emp
```

```
3  where sal not in (
4     (select min(sal) from emp),
5     (select max(sal) from emp)
6  )
```

DB2, Oracle, and SQL Server

Use an inline view with the windowing functions MAX OVER and MIN OVER to generate a result set from which you can easily eliminate the high and low values:

```
1 select avg(sal)
2   from (
3 select sal, min(sal)over() min_sal, max(sal)over() max_sal
4   from emp
5        ) x
6  where sal not in (min_sal,max_sal)
```

Discussion

MySQL and PostgreSQL

The subqueries return the highest and lowest salaries in the table. By using NOT IN against the values returned, you exclude the highest and lowest salaries from the average. Keep in mind that if there are duplicates (if multiple employees have the highest or lowest salaries), they will all be excluded from the average. If your goal is to exclude only a single instance of the high and low values, simply subtract them from the SUM and then divide:

```
select (sum(sal)-min(sal)-max(sal))/(count(*)-2)
  from emp
```

DB2, Oracle, and SQL Server

Inline view X returns each salary along with the highest and lowest salaries:

```
select sal, min(sal)over() min_sal, max(sal)over() max_sal
  from emp

      SAL   MIN_SAL   MAX_SAL
--------- --------- ---------
      800       800      5000
     1600       800      5000
     1250       800      5000
     2975       800      5000
     1250       800      5000
     2850       800      5000
     2450       800      5000
     3000       800      5000
     5000       800      5000
     1500       800      5000
     1100       800      5000
```

```
     950        800       5000
    3000        800       5000
    1300        800       5000
```

You can access the high and low salaries at every row, so finding which salaries are highest and/or lowest is trivial. The outer query filters the rows returned from inline view X such that any salary that matches either MIN_SAL or MAX_SAL is excluded from the average.

Robust Statistics

In statistical parlance, a mean calculated with the largest and smallest values removed is called a *trimmed mean*. This can be considered a safer estimate of the average, and is an example of a *robust statistic*, so called because they are less sensitive to problems such as bias. Recipe 7.16 is another example of a robust statistical tool. In both cases, these approaches are valuable to someone analyzing data within an RDBMS because they don't require the analyst to make assumptions that are difficult to test with the relatively limited range of statistical tools available in SQL.

7.14 Converting Alphanumeric Strings into Numbers

Problem

You have alphanumeric data and would like to return numbers only. You want to return the number 123321 from the string "paul123f321."

Solution

DB2

Use the functions TRANSLATE and REPLACE to extract numeric characters from an alphanumeric string:

```
1 select cast(
2        replace(
3      translate( 'paul123f321',
4                 repeat('#',26),
5                 'abcdefghijklmnopqrstuvwxyz'),'#','')
6        as integer ) as num
7   from t1
```

Oracle, SQL Server, and PostgreSQL

Use the functions TRANSLATE and REPLACE to extract numeric characters from an alphanumeric string:

```
1 select cast(
2        replace(
3      translate( 'paul123f321',
4                 'abcdefghijklmnopqrstuvwxyz',
5                 rpad('#',26,'#')),'#','')
6        as integer ) as num
7   from t1
```

MySQL

As of the time of this writing, MySQL doesn't support the TRANSLATE function; thus, a solution will not be provided.

Discussion

The only difference between the two solutions is syntax; DB2 uses the function REPEAT rather than RPAD, and the parameter list for TRANSLATE is in a different order. The following explanation uses the Oracle/PostgreSQL solution but is relevant to DB2 as well. If you run query inside out (starting with TRANSLATE only), you'll see this is simple. First, TRANSLATE converts any nonnumeric character to an instance of #:

```
select translate( 'paul123f321',
                  'abcdefghijklmnopqrstuvwxyz',
                  rpad('#',26,'#')) as num
  from t1

NUM
-----------
####123#321
```

Since all nonnumeric characters are now represented by #, simply use REPLACE to remove them, then use CAST the return the result as a number. This particular example is extremely simple because the data is alphanumeric. If additional characters can be stored, rather than fishing for those characters, it is easier to approach this problem differently: rather than finding nonnumeric characters and then removing them, find all numeric characters and remove anything that is not among them. The following example will help clarify this technique:

```
select replace(
     translate('paul123f321',
       replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#',''),
               rpad('#',length('paul123f321'),'#')),'#','') as num
  from t1

NUM
-----------
123321
```

This solution looks a bit more convoluted than the original but is not so bad once you break it down. Observe the innermost call to TRANSLATE:

```
select translate( 'paul123f321',
                  '0123456789',
                  rpad('#',10,'#'))
  from t1

TRANSLATE('
-----------
paul###f###
```

So, the initial approach is different; rather than replacing each nonnumeric character with an instance of #, you replace each numeric character with an instance of #. The next step removes all instances of #, thus leaving only nonnumeric characters:

```
select replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#','')
  from t1

REPLA
-----
paulf
```

The next step is to call TRANSLATE again, this time to replace each of the nonnumeric characters (from the previous query) with an instance of # in the original string:

```
select translate('paul123f321',
       replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#',''),
                rpad('#',length('paul123f321'),'#'))
  from t1

TRANSLATE('
-----------
####123#321
```

At this point, stop and examine the outermost call to TRANSLATE. The second parameter to RPAD (or the second parameter to REPEAT for DB2) is the length of the original string. This is convenient to use since no character can occur enough times to be greater than the string it is part of. Now that all nonnumeric characters are replaced by instances of #, the last step is to use REPLACE to remove all instances of #. Now you are left with a number.

7.15 Changing Values in a Running Total

Problem

You want to modify the values in a running total depending on the values in another column. Consider a scenario where you want to display the transaction history of a credit card account along with the current balance after each transaction. The following view, V, will be used in this example:

```
create view V (id,amt,trx)
as
select 1, 100, 'PR' from t1 union all
select 2, 100, 'PR' from t1 union all
select 3, 50,  'PY' from t1 union all
select 4, 100, 'PR' from t1 union all
select 5, 200, 'PY' from t1 union all
select 6, 50,  'PY' from t1

  select * from V

ID         AMT  TR
--  ----------  --
 1         100  PR
 2         100  PR
 3          50  PY
 4         100  PR
 5         200  PY
 6          50  PY
```

The ID column uniquely identifies each transaction. The AMT column represents the amount of money involved in each transaction (either a purchase or a payment). The TRX column defines the type of transaction; a payment is "PY" and a purchase is "PR." If the value for TRX is PY, you want the current value for AMT subtracted from the running total; if the value for TRX is PR, you want the current value for AMT added to the running total. Ultimately you want to return the following result set:

```
TRX_TYPE        AMT    BALANCE
-------- ---------- ----------
PURCHASE        100        100
PURCHASE        100        200
PAYMENT          50        150
PURCHASE        100        250
PAYMENT         200         50
PAYMENT          50          0
```

Solution

Use the window function SUM OVER to create the running total along with a CASE expression to determine the type of transaction:

```
 1 select case when trx = 'PY'
 2             then 'PAYMENT'
 3             else 'PURCHASE'
 4         end trx_type,
 5         amt,
 6         sum(
 7          case when trx = 'PY'
 8             then -amt else amt
 9          end
10        ) over (order by id,amt) as balance
11 from V
```

Discussion

The CASE expression determines whether the current AMT is added or deducted from the running total. If the transaction is a payment, the AMT is changed to a negative value, thus reducing the amount of the running total. The result of the CASE expression is shown here:

```
select case when trx = 'PY'
            then 'PAYMENT'
            else 'PURCHASE'
       end trx_type,
       case when trx = 'PY'
            then -amt else amt
       end as amt
  from V

TRX_TYPE       AMT
-------- ---------
PURCHASE       100
PURCHASE       100
PAYMENT        -50
PURCHASE       100
PAYMENT       -200
PAYMENT        -50
```

After evaluating the transaction type, the values for AMT are then added to or subtracted from the running total. For an explanation on how the window function, SUM OVER, or the scalar subquery creates the running total, see recipe Recipe 7.6.

7.16 Finding Outliers Using the Median Absolute Deviation

Problem

You want to identify values in your data that may be suspect. There are various reasons why values could be suspect—there could be a data collection issue, such as an error with the meter that records the value. There could be a data entry error such as

a typo or similar. There could also be unusual circumstances when the data was generated that mean the data point is correct, but they still require you to use caution in any conclusion you make from the data. Therefore, you want to detect outliers.

A common way to detect outliers, taught in many statistics courses aimed at non-statisticians, is to calculate the standard deviation of the data and decide that data points more than three standard deviations (or some other similar distance) are outliers. However, this method can misidentify outliers if the data don't follow a normal distribution, especially if the spread of data isn't symmetrical or doesn't thin out in the same way as a normal distribution as you move further from the mean.

Solution

First find the median of the values using the recipe for finding the median from earlier in this chapter. You will need to put this query into a CTE to make it available for further querying. The deviation is the absolute difference between the median and each value; the median absolute deviation is the median of this value, so we need to calculate the median again.

SQL Server

SQL Server has the PERCENTILE_CONT function, which simplifies finding the median. As we need to find two different medians and manipulate them, we need a series of CTEs:

```
with median (median)
as
(select distinct percentile_cont(0.5) within group(order by sal)
        over()
from emp),

Deviation (Deviation)
  as
(Select abs(sal-median)
from emp join median on 1=1),

MAD (MAD) as
(select DISTINCT PERCENTILE_CONT(0.5) within group(order by deviation) over()
from Deviation )

select abs(sal-MAD)/MAD, sal, ename, job
from MAD join emp on 1=1
```

PostgreSQL and DB2

The overall pattern is the same, but there is different syntax for PERCENTILE_CONT, as PostgreSQL and DB2 treat PERCENTILE_CONT as an aggregate function rather than strictly a window function:

```
with median (median)
as
(select percentile_cont(0.5) within group(order by sal)
from emp),

devtab (deviation)
  as
(select abs(sal-median)
from emp join median),

MedAbsDeviation (MAD) as
(select percentile_cont (0.5) within group(order by deviation)
from devtab)

select abs(sal-MAD)/MAD, sal, ename, job
FROM MedAbsDeviation join emp
```

Oracle

The recipe is simplified for Oracle users due to the existence of a median function. However, we still need to use a CTE to handle the scalar value of deviation:

```
with
Deviation (Deviation)
  as
(select abs(sal-median(sal))
from emp),

MAD (MAD) as
(select median(Deviation)
from Deviation )

select abs(sal-MAD)/MAD, sal, ename, job
FROM MAD join emp
```

MySQL

As we saw in the earlier section on the median, there is unfortunately no MEDIAN or PERCENTILE_CONT function in MySQL. This means that each of the medians we need to find to compute the median absolute deviation is two subqueries within a CTE. This makes the MySQL a little long-winded:

```
with rank_tab (sal, rank_sal) as (
select sal, cume_dist() over (order by sal)
from emp),
inter as
(
select sal, rank_sal from rank_tab
where rank_sal>=0.5
union
select sal, rank_sal from rank_tab
where rank_sal<=0.5
```

```
)
,

medianSal (medianSal) as

(
select (max(sal)+min(sal))/2
from inter),
deviationSal (Sal,deviationSal) as
(select Sal,abs(sal-medianSal)
from emp join medianSal
on 1=1
)
,

distDevSal (sal,deviationSal,distDeviationSal) as

(
select sal,deviationSal,cume_dist() over (order by deviationSal)
from deviationSal
),

DevInter (DevInter, sal) as
(
select min(deviationSal), sal
from distDevSal
where distDeviationSal >= 0.5

union

select max(DeviationSal), sal
from distDevSal
where distDeviationSal <= 0.5
),

MAD (MedianAbsoluteDeviance) as
(
select abs(emp.sal-(min(devInter)+max(devInter))/2)
from emp join DevInter on 1=1
)

select emp.sal,MedianAbsoluteDeviance,
(emp.sal-deviationSal)/MedianAbsoluteDeviance
from (emp join MAD on 1=1)
         join deviationSal on emp.sal=deviationSal.sal
```

Discussion

In each case the recipe follows a similar strategy. First we need to calculate the median, and then we need to calculate the median of the difference between each value and the median, which is the actual median absolute deviation. Finally, we need

to use a query to find the ratio of the deviation of each value to the median deviation. At that point, we can use the outcome in a similar way to the standard deviation. For example, if a value is three or more deviations from the median, it can be considered an outlier, to use a common interpretation.

As mentioned earlier, the benefit of this approach over the standard deviation is that the interpretation is still valid even if the data doesn't display a normal distribution. For example, it can be lopsided, and the median absolute deviation will still give a sound answer.

In our salary data, there is one salary that is more than three absolute deviations from the median: the CEO's.

Although there are differing opinions about the fairness of CEO salaries versus those of most other workers, given that the outlier salary belongs to the CEO, it fits with our understanding of the data. In other contexts, if there wasn't a clear explanation of why the value differed so much, it could lead us to question whether that value was correct or whether the value made sense when taken with the rest of the values (e.g., if it not actually an error, it might make us think we need to analyze our data within more than one subgroup).

Many of the common statistics, such as the mean and the standard deviation, assume that the shape of the data is a bell curve—a normal distribution. This is true for many data sets, and also not true for many data sets.

There are a number of methods for testing whether a data set follows a normal distribution, both by visualizing the data and through calculations. Statistical packages commonly contain functions for these tests, but they are nonexistent and hard to replicate in SQL. However, there are often alternative statistical tools that don't assume the data takes a particular form—nonparametric statistics—and these are safer to use.

7.17 Finding Anomalies Using Benford's Law

Problem

Although outliers, as shown in the previous recipe, are a readily identifiable form of anomalous data, some other data is less easy to identify as problematic. One way to detect situations where there are anomalous data but no obvious outliers is to look at the frequency of digits, which is usually expected to follow Benford's law. Although using Benford's law is most often associated with detecting fraud in situations where humans have added fake numbers to a data set, it can be used more generally to

detect data that doesn't follow expected patterns. For example, it can detect errors such as duplicated data points, which won't necessarily stand out as outliers.

Solution

To use Benford's law, you need to calculate the expected distribution of digits and then the actual distribution to compare. Although the most sophisticated uses look at first, second, and combinations of digits, in this example we will stick to just the first digits.

You compare the frequency predicted by Benford's law with the actual frequency of your data. Ultimately you want four columns—the first digit, the count of how many times each first digit appears, the frequency of first digits predicted by Benford's law, and the actual frequency:

```
with
FirstDigits (FirstDigit)
as
(select left(cast(SAL as CHAR),1) as FirstDigit
        from emp),

TotalCount (Total)
as
 (select count(*)
  from emp),

ExpectedBenford (Digit,Expected)
as
  (select value,(log10(value + 1) - log10(value)) as expected
   from t10
   where value < 10)

select count(FirstDigit),Digit
,coalesce(count(*)/Total,0) as ActualProportion,Expected
From FirstDigits
     Join TotalCount
     Right Join ExpectedBenford
     on FirstDigits.FirstDigit=ExpectedBenford.Digit
group by Digit
order by Digit;
```

Discussion

Because we need to make use of two different counts—one of the total rows, and another of the number of rows containing each different first digit—we need to use a CTE. Strictly speaking, we don't need to put the expected Benford's law results into a separate query within the CTE, but we have done so in this case as it allows us to identify the digits with a zero count and display them in the table via the right join.

It's also possible to produce the FirstDigits count in the main query, but we have chosen not to improve readability through not needing to repeat the LEFT(CAST… expression in the GROUP BY clause.

The math behind Benford's law is simple:

$$\text{Expected frequency} = \log_{10}\left(\frac{d+1}{d}\right)$$

We can use the T10 pivot table to generate the appropriate values. From there we just need to calculate the actual frequencies for comparison, which first requires us to identify the first digit.

Benford's law works best when there is a relatively large collection of values to apply it to, and when those values span more than one order of magnitude (10, 100, 1,000, etc.). Those conditions aren't entirely met here. At the same time, the deviation from expected should still make us suspicious that these values are in some sense made-up values and worth investigating further.

7.18 Summing Up

An enterprise's data is frequently found in a database supported by SQL, so it makes sense to use SQL to try to understand that data. SQL doesn't have the full array of statistical tools you would expect in a purpose-built package such as SAS, the statistical programming language R, or Python's statistical libraries. However, it does have a rich set of tools for calculation that as we have seen can provide a deep understanding of the statistical properties of your data.

CHAPTER 8

Date Arithmetic

This chapter introduces techniques for performing simple date arithmetic. Recipes cover common tasks such as adding days to dates, finding the number of business days between dates, and finding the difference between dates in days.

Being able to successfully manipulate dates with your RDBMS's built-in functions can greatly improve your productivity. For all the recipes in this chapter, we try to take advantage of each RDBMS's built-in functions. In addition, we have chosen to use one date format for all the recipes, DD-MON-YYYY. Of course, there are a number of other commonly used formats, such as DD-MM-YYYY, the ISO standard format.

We chose to standardize on DD-MON-YYYY to benefit those of you who work with one RDBMS and want to learn others. Seeing one standard format will help you focus on the different techniques and functions provided by each RDBMS without having to worry about default date formats.

This chapter focuses on basic date arithmetic. You'll find more advanced date recipes in the following chapter. The recipes presented in this chapter use simple date data types. If you are using more complex date data types, you will need to adjust the solutions accordingly.

8.1 Adding and Subtracting Days, Months, and Years

Problem

You need to add or subtract some number of days, months, or years from a date. For example, using the HIREDATE for employee CLARK, you want to return six different dates: five days before and after CLARK was hired, five months before and after

CLARK was hired, and, finally, five years before and after CLARK was hired. CLARK was hired on 09-JUN-2006, so you want to return the following result set:

```
HD_MINUS_5D HD_PLUS_5D  HD_MINUS_5M HD_PLUS_5M  HD_MINUS_5Y HD_PLUS_5Y
----------- ----------- ----------- ----------- ----------- -----------
04-JUN-2006 14-JUN-2006 09-JAN-2006 09-NOV-2006 09-JUN-2001 09-JUN-2001
12-NOV-2006 22-NOV-2006 17-JUN-2006 17-APR-2007 17-NOV-2001 17-NOV-2001
18-JAN-2007 28-JAN-2007 23-AUG-2006 23-JUN-2007 23-JAN-2002 23-JAN-2002
```

Solution

DB2

Standard addition and subtraction is allowed on date values, but any value that you add to or subtract from a date must be followed by the unit of time it represents:

```
1 select hiredate -5 day   as hd_minus_5D,
2        hiredate +5 day   as hd_plus_5D,
3        hiredate -5 month as hd_minus_5M,
4        hiredate +5 month as hd_plus_5M,
5        hiredate -5 year  as hd_minus_5Y,
6        hiredate +5 year  as hd_plus_5Y
7   from emp
8  where deptno = 10
```

Oracle

Use standard addition and subtraction for days, and use the ADD_MONTHS function to add and subtract months and years:

```
1 select hiredate-5                as hd_minus_5D,
2        hiredate+5                as hd_plus_5D,
3        add_months(hiredate,-5)   as hd_minus_5M,
4        add_months(hiredate,5)    as hd_plus_5M,
5        add_months(hiredate,-5*12) as hd_minus_5Y,
6        add_months(hiredate,5*12)  as hd_plus_5Y
7   from emp
8  where deptno = 10
```

PostgreSQL

Use standard addition and subtraction with the INTERVAL keyword specifying the unit of time to add or subtract. Single quotes are required when specifying an INTERVAL value:

```
1 select hiredate - interval '5 day'   as hd_minus_5D,
2        hiredate + interval '5 day'   as hd_plus_5D,
3        hiredate - interval '5 month' as hd_minus_5M,
4        hiredate + interval '5 month' as hd_plus_5M,
5        hiredate - interval '5 year'  as hd_minus_5Y,
6        hiredate + interval '5 year'  as hd_plus_5Y
```

```
  from emp
 where deptno=10
```

MySQL

Use standard addition and subtraction with the INTERVAL keyword specifying the unit of time to add or subtract. Unlike the PostgreSQL solution, you do not place single quotes around the INTERVAL value:

```
select hiredate - interval 5 day   as hd_minus_5D,
       hiredate + interval 5 day   as hd_plus_5D,
       hiredate - interval 5 month as hd_minus_5M,
       hiredate + interval 5 month as hd_plus_5M,
       hiredate - interval 5 year  as hd_minus_5Y,
       hiredate + interval 5 year  as hd_plus_5Y
  from emp
 where deptno=10
```

Alternatively, you can use the DATE_ADD function, which is shown here:

```
select date_add(hiredate,interval -5 day)   as hd_minus_5D,
       date_add(hiredate,interval  5 day)   as hd_plus_5D,
       date_add(hiredate,interval -5 month) as hd_minus_5M,
       date_add(hiredate,interval  5 month) as hd_plus_5M,
       date_add(hiredate,interval -5 year)  as hd_minus_5Y,
       date_add(hiredate,interval  5 year)  as hd_plus_5DY
  from emp
 where deptno=10
```

SQL Server

Use the DATEADD function to add or subtract different units of time to/from a date:

```
select dateadd(day,-5,hiredate)   as hd_minus_5D,
       dateadd(day,5,hiredate)    as hd_plus_5D,
       dateadd(month,-5,hiredate) as hd_minus_5M,
       dateadd(month,5,hiredate)  as hd_plus_5M,
       dateadd(year,-5,hiredate)  as hd_minus_5Y,
       dateadd(year,5,hiredate)   as hd_plus_5Y
  from emp
 where deptno = 10
```

Discussion

The Oracle solution takes advantage of the fact that integer values represent days when performing date arithmetic. However, that's true only of arithmetic with DATE types. Oracle also has TIMESTAMP types. For those, you should use the INTERVAL solution shown for PostgreSQL. Beware too, of passing TIMESTAMPs to old-style date functions such as ADD_MONTHS. By doing so, you can lose any fractional seconds that such TIMESTAMP values may contain.

The INTERVAL keyword and the string literals that go with it represent ISO-standard SQL syntax. The standard requires that interval values be enclosed within single quotes. PostgreSQL (and Oracle9*i* Database and later) complies with the standard. MySQL deviates somewhat by omitting support for the quotes.

8.2 Determining the Number of Days Between Two Dates

Problem

You want to find the difference between two dates and represent the result in days. For example, you want to find the difference in days between the HIREDATEs of employee ALLEN and employee WARD.

Solution

DB2

Use two inline views to find the HIREDATEs for WARD and ALLEN. Then subtract one HIREDATE from the other using the DAYS function:

```
 1 select days(ward_hd) - days(allen_hd)
 2   from (
 3 select hiredate as ward_hd
 4   from emp
 5  where ename = 'WARD'
 6        ) x,
 7        (
 8 select hiredate as allen_hd
 9   from emp
10  where ename = 'ALLEN'
11        ) y
```

Oracle and PostgreSQL

Use two inline views to find the HIREDATEs for WARD and ALLEN, and then subtract one date from the other:

```
 1 select ward_hd - allen_hd
 2   from (
 3 select hiredate as ward_hd
 4   from emp
 5  where ename = 'WARD'
 6        ) x,
 7        (
 8 select hiredate as allen_hd
 9   from emp
10  where ename = 'ALLEN'
11        ) y
```

MySQL and SQL Server

Use the function DATEDIFF to find the number of days between two dates. MySQL's version of DATEDIFF requires only two parameters (the two dates you want to find the difference in days between), and the smaller of the two dates should be passed first to avoid negative values (opposite in SQL Server). SQL Server's version of the function allows you to specify what you want the return value to represent (in this example you want to return the difference in days). The solution following uses the SQL Server version:

```
 1 select datediff(day,allen_hd,ward_hd)
 2   from (
 3 select hiredate as ward_hd
 4   from emp
 5  where ename = 'WARD'
 6        ) x,
 7        (
 8 select hiredate as allen_hd
 9   from emp
10  where ename = 'ALLEN'
11        ) y
```

MySQL users can simply remove the first argument of the function and flip-flop the order in which ALLEN_HD and WARD_HD is passed.

Discussion

For all solutions, inline views X and Y return the HIREDATEs for employees WARD and ALLEN, respectively. For example:

```
select ward_hd, allen_hd
    from (
select hiredate as ward_hd
  from emp
 where ename = 'WARD'
       ) y,
       (
select hiredate as allen_hd
  from emp
 where ename = 'ALLEN'
       ) x

WARD_HD     ALLEN_HD
----------- ----------
22-FEB-2006 20-FEB-2006
```

You'll notice a Cartesian product is created, because there is no join specified between X and Y. In this case, the lack of a join is harmless as the cardinalities for X and Y are both 1; thus, the result set will ultimately have one row (obviously, because $1 \times 1 = 1$).

To get the difference in days, simply subtract one of the two values returned from the other using methods appropriate for your database.

8.3 Determining the Number of Business Days Between Two Dates

Problem

Given two dates, you want to find how many "working" days are between them, including the two dates themselves. For example, if January 10th is a Tuesday and January 11th is a Monday, then the number of working days between these two dates is two, as both days are typical workdays. For this recipe, a "business day" is defined as any day that is not Saturday or Sunday.

Solution

The solution examples find the number of business days between the HIREDATEs of BLAKE and JONES. To determine the number of business days between two dates, you can use a pivot table to return a row for each day between the two dates (including the start and end dates). Having done that, finding the number of business days is simply counting the dates returned that are not Saturday or Sunday.

If you want to exclude holidays as well, you can create a HOLIDAYS table. Then include a simple NOT IN predicate to exclude days listed in HOLIDAYS from the solution.

DB2

Use the pivot table T500 to generate the required number of rows (representing days) between the two dates. Then count each day that is not a weekend. Use the DAYNAME function to return the weekday name of each date. For example:

```
select sum(case when dayname(jones_hd+t500.id day -1 day)
                  in ( 'Saturday','Sunday' )
                then 0 else 1
           end) as days
  from (
select max(case when ename = 'BLAKE'
                then hiredate
           end) as blake_hd,
       max(case when ename = 'JONES'
                then hiredate
           end) as jones_hd
  from emp
```

```
 13  where ename in ( 'BLAKE','JONES' )
 14        ) x,
 15        t500
 16  where t500.id <= blake_hd-jones_hd+1
```

MySQL

Use the pivot table T500 to generate the required number of rows (days) between the two dates. Then count each day that is not a weekend. Use the DATE_ADD function to add days to each date. Use the DATE_FORMAT function to obtain the weekday name of each date:

```
 1 select sum(case when date_format(
 2                       date_add(jones_hd,
 3                                interval t500.id-1 DAY),'%a')
 4                  in ( 'Sat','Sun' )
 5                then 0 else 1
 6            end) as days
 7   from (
 8 select max(case when ename = 'BLAKE'
 9                then hiredate
10            end) as blake_hd,
11        max(case when ename = 'JONES'
12                then hiredate
13             end) as jones_hd
14   from emp
15  where ename in ( 'BLAKE','JONES' )
16        ) x,
17        t500
18  where t500.id <= datediff(blake_hd,jones_hd)+1
```

Oracle

Use the pivot table T500 to generate the required number of rows (days) between the two dates, and then count each day that is not a weekend. Use the TO_CHAR function to obtain the weekday name of each date:

```
 1 select sum(case when to_char(jones_hd+t500.id-1,'DY')
 2                  in ( 'SAT','SUN' )
 3                then 0 else 1
 4            end) as days
 5   from (
 6 select max(case when ename = 'BLAKE'
 7                then hiredate
 8            end) as blake_hd,
 9        max(case when ename = 'JONES'
10                then hiredate
11            end) as jones_hd
12   from emp
13  where ename in ( 'BLAKE','JONES' )
14        ) x,
```

```
       t500
 where t500.id <= blake_hd-jones_hd+1
```

PostgreSQL

Use the pivot table T500 to generate the required number of rows (days) between the two dates. Then count each day that is not a weekend. Use the TO_CHAR function to obtain the weekday name of each date:

```
select sum(case when trim(to_char(jones_hd+t500.id-1,'DAY'))
                  in ( 'SATURDAY','SUNDAY' )
                then 0 else 1
           end) as days
  from (
select max(case when ename = 'BLAKE'
                then hiredate
           end) as blake_hd,
       max(case when ename = 'JONES'
                then hiredate
           end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
       ) x,
       t500
 where t500.id <= blake_hd-jones_hd+1
```

SQL Server

Use the pivot table T500 to generate the required number of rows (days) between the two dates, and then count each day that is not a weekend. Use the DATENAME function to obtain the weekday name of each date:

```
select sum(case when datename(dw,jones_hd+t500.id-1)
                  in ( 'SATURDAY','SUNDAY' )
                 then 0 else 1
           end) as days
  from (
selectmax(case when ename = 'BLAKE'
                then hiredate
           end) as blake_hd,
       max(case when ename = 'JONES'
                then hiredate
           end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
       ) x,
       t500
 where t500.id <= datediff(day,jones_hd-blake_hd)+1
```

Discussion

While each RDBMS requires the use of different built-in functions to determine the name of a day, the overall solution approach is the same for each. The solution can be broken into two steps:

1. Return the days between the start date and end date (inclusive).
2. Count how many days (i.e., rows) there are, excluding weekends.

Inline view X performs step one. If you examine inline view X, you'll notice the use of the aggregate function MAX, which the recipe uses to remove NULLs. If the use of MAX is unclear, the following output might help you understand. The output shows the results from inline view X without MAX:

```
select case when ename = 'BLAKE'
            then hiredate
       end as blake_hd,
       case when ename = 'JONES'
            then hiredate
       end as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )

BLAKE_HD    JONES_HD
----------- -----------
            02-APR-2006
01-MAY-2006
```

Without MAX, two rows are returned. By using MAX you return only one row instead of two, and the NULLs are eliminated:

```
select max(case when ename = 'BLAKE'
            then hiredate
       end) as blake_hd,
       max(case when ename = 'JONES'
            then hiredate
       end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )

BLAKE_HD    JONES_HD
----------- -----------
01-MAY-2006 02-APR-2006
```

The number of days (inclusive) between the two dates here is 30. Now that the two dates are in one row, the next step is to generate one row for each of those 30 days. To return the 30 days (rows), use table T500. Since each value for ID in table T500 is simply one greater than the one before it, add each row returned by T500 to the earlier of the two dates (JONES_HD) to generate consecutive days starting from

JONES_HD up to and including BLAKE_HD. The result of this addition is shown here (using Oracle syntax):

```
select x.*, t500.*, jones_hd+t500.id-1
  from (
select max(case when ename = 'BLAKE'
                then hiredate
           end) as blake_hd,
       max(case when ename = 'JONES'
                then hiredate
           end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
       ) x,
       t500
 where t500.id <= blake_hd-jones_hd+1

BLAKE_HD    JONES_HD            ID JONES_HD+T5
----------- ----------- ---------- -----------
01-MAY-2006 02-APR-2006          1 02-APR-2006
01-MAY-2006 02-APR-2006          2 03-APR-2006
01-MAY-2006 02-APR-2006          3 04-APR-2006
01-MAY-2006 02-APR-2006          4 05-APR-2006
01-MAY-2006 02-APR-2006          5 06-APR-2006
01-MAY-2006 02-APR-2006          6 07-APR-2006
01-MAY-2006 02-APR-2006          7 08-APR-2006
01-MAY-2006 02-APR-2006          8 09-APR-2006
01-MAY-2006 02-APR-2006          9 10-APR-2006
01-MAY-2006 02-APR-2006         10 11-APR-2006
01-MAY-2006 02-APR-2006         11 12-APR-2006
01-MAY-2006 02-APR-2006         12 13-APR-2006
01-MAY-2006 02-APR-2006         13 14-APR-2006
01-MAY-2006 02-APR-2006         14 15-APR-2006
01-MAY-2006 02-APR-2006         15 16-APR-2006
01-MAY-2006 02-APR-2006         16 17-APR-2006
01-MAY-2006 02-APR-2006         17 18-APR-2006
01-MAY-2006 02-APR-2006         18 19-APR-2006
01-MAY-2006 02-APR-2006         19 20-APR-2006
01-MAY-2006 02-APR-2006         20 21-APR-2006
01-MAY-2006 02-APR-2006         21 22-APR-2006
01-MAY-2006 02-APR-2006         22 23-APR-2006
01-MAY-2006 02-APR-2006         23 24-APR-2006
01-MAY-2006 02-APR-2006         24 25-APR-2006
01-MAY-2006 02-APR-2006         25 26-APR-2006
01-MAY-2006 02-APR-2006         26 27-APR-2006
01-MAY-2006 02-APR-2006         27 28-APR-2006
01-MAY-2006 02-APR-2006         28 29-APR-2006
01-MAY-2006 02-APR-2006         29 30-APR-2006
01-MAY-2006 02-APR-2006         30 01-MAY-2006
```

If you examine the WHERE clause, you'll notice that you add 1 to the difference between BLAKE_HD and JONES_HD to generate the required 30 rows (otherwise, you would get 29 rows). You'll also notice that you subtract 1 from T500.ID in the SELECT list of the outer query, since the values for ID start at 1 and adding 1 to JONES_HD would cause JONES_HD to be excluded from the final count.

Once you generate the number of rows required for the result set, use a CASE expression to "flag" whether each of the days returned is weekday or weekend (return a 1 for a weekday and a 0 for a weekend). The final step is to use the aggregate function SUM to tally up the number of 1s to get the final answer.

8.4 Determining the Number of Months or Years Between Two Dates

Problem

You want to find the difference between two dates in terms of either months or years. For example, you want to find the number of months between the first and last employees hired, and you also want to express that value as some number of years.

Solution

Since there are always 12 months in a year, you can find the number of months between 2 dates and then divide by 12 to get the number of years. After getting comfortable with the solution, you'll want to round the results up or down depending on what you want for the year. For example, the first HIREDATE in table EMP is 17-DEC-1980 and the last is 12-JAN-1983. If you do the math on the years (1983 minus 1980), you get 3 years, yet the difference in months is approximately 25 (a little over 2 years). You should tweak the solution as you see fit. The following solutions will return 25 months and approximately 2 years.

DB2 and MySQL

Use the functions YEAR and MONTH to return the four-digit year and the two-digit month for the dates supplied:

```
select mnth, mnth/12
  from (
select (year(max_hd) - year(min_hd))*12 +
       (month(max_hd) - month(min_hd)) as mnth
  from (
select min(hiredate) as min_hd, max(hiredate) as max_hd
  from emp
       ) x
       ) y
```

Oracle

Use the function MONTHS_BETWEEN to find the difference between two dates in months (to get years, simply divide by 12):

```
1 select months_between(max_hd,min_hd),
2        months_between(max_hd,min_hd)/12
3   from (
4 select min(hiredate) min_hd, max(hiredate) max_hd
5   from emp
6        ) x
```

PostgreSQL

Use the function EXTRACT to return the four-digit year and two-digit month for the dates supplied:

```
 1 select mnth, mnth/12
 2   from (
 3 select ( extract(year from max_hd)
 4          extract(year from min_hd) ) * 12
 5        +
 6        ( extract(month from max_hd)
 7          extract(month from min_hd) ) as mnth
 8   from (
 9 select min(hiredate) as min_hd, max(hiredate) as max_hd
10   from emp
11        ) x
12        ) y
```

SQL Server

Use the function DATEDIFF to find the difference between two dates, and use the DATEPART argument to specify months and years as the time units returned:

```
1 select datediff(month,min_hd,max_hd),
2        datediff(year,min_hd,max_hd)
3   from (
4 select min(hiredate) min_hd, max(hiredate) max_hd
5   from emp
6        ) x
```

Discussion

DB2, MySQL, and PostgreSQL

Once you extract the year and month for MIN_HD and MAX_HD in the PostgreSQL solution, the method for finding the months and years between MIN_HD and MAX_HD is the same for all three RDBMs. This discussion will cover all three solutions.

Inline view X returns the earliest and latest HIREDATEs in table EMP and is shown here:

```
select min(hiredate) as min_hd,
       max(hiredate) as max_hd
  from emp

MIN_HD      MAX_HD
----------- -----------
17-DEC-1980 12-JAN-1983
```

To find the months between MAX_HD and MIN_HD, multiply the difference in years between MIN_HD and MAX_HD by 12, and then add the difference in months between MAX_HD and MIN_HD. If you are having trouble seeing how this works, return the date component for each date. The numeric values for the years and months are shown here:

```
select year(max_hd) as max_yr, year(min_hd) as min_yr,
       month(max_hd) as max_mon, month(min_hd) as min_mon
  from (
select min(hiredate) as min_hd, max(hiredate) as max_hd
  from emp
       ) x

MAX_YR     MIN_YR    MAX_MON    MIN_MON
------ ---------- ---------- ----------
  1983       1980          1         12
```

Looking at these results, finding the months between MAX_HD and MIN_HD is simply (1983 – 1980) × 12 + (1 – 12). To find the number of years between MIN_HD and MAX_HD, divide the number of months by 12. Again, depending on the results you are looking for, you will want to round the values.

Oracle and SQL Server

Inline view X returns the earliest and latest HIREDATEs in table EMP and is shown here:

```
select min(hiredate) as min_hd, max(hiredate) as max_hd
  from emp

MIN_HD      MAX_HD
----------- -----------
17-DEC-1980 12-JAN-1983
```

The functions supplied by Oracle and SQL Server (MONTHS_BETWEEN and DATEDIFF, respectively) will return the number of months between two given dates. To find the year, divide the number of months by 12.

8.5 Determining the Number of Seconds, Minutes, or Hours Between Two Dates

Problem

You want to return the difference in seconds between two dates. For example, you want to return the difference between the HIREDATEs of ALLEN and WARD in seconds, minutes, and hours.

Solution

If you can find the number of days between two dates, you can find seconds, minutes, and hours as they are the units of time that make up a day.

DB2

Use the function DAYS to find the difference between ALLEN_HD and WARD_HD in days. Then multiply to find each unit of time:

```
select dy*24 hr, dy*24*60 min, dy*24*60*60 sec
  from (
select ( days(max(case when ename = 'WARD'
                  then hiredate
             end)) -
         days(max(case when ename = 'ALLEN'
                  then hiredate
             end))
       ) as dy
  from emp
       ) x
```

MySQL

Use the DATEDIFF function to return the number of days between ALLEN_HD and WARD_HD. Then multiply to find each unit of time:

```
select datediff(day,allen_hd,ward_hd)*24 hr,
       datediff(day,allen_hd,ward_hd)*24*60 min,
       datediff(day,allen_hd,ward_hd)*24*60*60 sec
  from (
select max(case when ename = 'WARD'
                 then hiredate
           end) as ward_hd,
       max(case when ename = 'ALLEN'
                then hiredate
           end) as allen_hd
  from emp
       ) x
```

SQL Server

Use the DATEDIFF function to return the number of days between ALLEN_HD and WARD_HD. Then use the DATEPART argument to specify the required time unit:

```
 1 select datediff(day,allen_hd,ward_hd,hour) as hr,
 2        datediff(day,allen_hd,ward_hd,minute) as min,
 3        datediff(day,allen_hd,ward_hd,second) as sec
 4   from (
 5 select max(case when ename = 'WARD'
 6                  then hiredate
 7            end) as ward_hd,
 8        max(case when ename = 'ALLEN'
 9                 then hiredate
10            end) as allen_hd
11   from emp
12        ) x
```

Oracle and PostgreSQL

Use subtraction to return the number of days between ALLEN_HD and WARD_ HD. Then multiply to find each unit of time:

```
 1 select dy*24 as hr, dy*24*60 as min, dy*24*60*60 as sec
 2   from (
 3 select (max(case when ename = 'WARD'
 4                 then hiredate
 5            end) -
 6        max(case when ename = 'ALLEN'
 7                 then hiredate
 8            end)) as dy
 9   from emp
10        ) x
```

Discussion

Inline view X for all solutions returns the HIREDATEs for WARD and ALLEN, as shown here:

```
select max(case when ename = 'WARD'
                 then hiredate
           end) as ward_hd,
       max(case when ename = 'ALLEN'
                 then hiredate
           end) as allen_hd
  from emp

WARD_HD     ALLEN_HD
----------- -----------
22-FEB-2006 20-FEB-2006
```

Multiply the number of days between WARD_HD and ALLEN_HD by 24 (hours in a day), 1440 (minutes in a day), and 86400 (seconds in a day).

8.6 Counting the Occurrences of Weekdays in a Year

Problem

You want to count the number of times each weekday occurs in one year.

Solution

To find the number of occurrences of each weekday in a year, you must:

1. Generate all possible dates in the year.
2. Format the dates such that they resolve to the name of their respective weekdays.
3. Count the occurrence of each weekday name.

DB2

Use recursive WITH to avoid the need to SELECT against a table with at least 366 rows. Use the function DAYNAME to obtain the weekday name for each date, and then count the occurrence of each:

```
with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (current_date
        dayofyear(current_date) day)
        +1 day as start_date
  from t1
       ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
)
select dayname(start_date),count(*)
  from x
 group by dayname(start_date)
```

MySQL

Select against table T500 to generate enough rows to return every day in the year. Use the DATE_FORMAT function to obtain the weekday name of each date, and then count the occurrence of each name:

```
select date_format(
          date_add(
              cast(
            concat(year(current_date),'-01-01')
                   as date),
                   interval t500.id-1 day),
                   '%W') day,
       count(*)
  from t500
 where t500.id <= datediff(
                         cast(
                       concat(year(current_date)+1,'-01-01')
                              as date),
                         cast(
                       concat(year(current_date),'-01-01')
                              as date))
group by date_format(
            date_add(
                cast(
              concat(year(current_date),'-01-01')
                     as date),
                     interval t500.id-1 day),
                     '%W')
```

Oracle

You can use the recursive CONNECT BY to return each day in a year:

```
with x as (
select level lvl
  from dual
 connect by level <= (
   add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
 )
)
select to_char(trunc(sysdate,'y')+lvl-1,'DAY'), count(*)
  from x
 group by to_char(trunc(sysdate,'y')+lvl-1,'DAY')
```

PostgreSQL

Use the built-in function GENERATE_SERIES to generate one row for every day in the year. Then use the TO_CHAR function to obtain the weekday name of each date. Finally, count the occurrence of each weekday name. For example:

```
select to_char(
          cast(
    date_trunc('year',current_date)
               as date) + gs.id-1,'DAY'),
       count(*)
  from generate_series(1,366) gs(id)
 where gs.id <= (cast
```

```
                      ( date_trunc('year',current_date) +
                           interval '12 month' as date) -
cast(date_trunc('year',current_date)
                       as date))
 group by to_char(
             cast(
       date_trunc('year',current_date)
          as date) + gs.id-1,'DAY')
```

SQL Server

Use the recursive WITH to avoid the need to SELECT against a table with at least 366 rows. Use the DATENAME function to obtain the weekday name of each date, and then count the occurrence of each name. For example:

```
with x (start_date,end_date)
as (
select start_date,
       dateadd(year,1,start_date) end_date
  from (
select cast(
       cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
  from t1
       ) tmp
union all
select dateadd(day,1,start_date), end_date
  from x
 where dateadd(day,1,start_date) < end_date
)
select datename(dw,start_date),count(*)
  from x
 group by datename(dw,start_date)
OPTION (MAXRECURSION 366)
```

Discussion

DB2

Inline view TMP, in the recursive WITH view X, returns the first day of the current year and is shown here:

```
select (current_date
        dayofyear(current_date) day)
        +1 day as start_date
  from t1

START_DATE
-------------
01-JAN-2005
```

The next step is to add one year to START_DATE so that you have the beginning and end dates. You need to know both because you want to generate every day in a year. START_DATE and END_DATE are shown here:

```
select start_date,
       start_date + 1 year end_date
  from (
select (current_date
        dayofyear(current_date) day)
        +1 day as start_date
  from t1
       ) tmp

        START_DATE  END_DATE
        ----------- ------------
        01-JAN-2005 01-JAN-2006
```

The next step is to recursively increment START_DATE by one day, stopping before it equals END_DATE. A portion of the rows returned by the recursive view X is shown here:

```
with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (current_date -
        dayofyear(current_date) day)
        +1 day as start_date
  from t1
       ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
)
select * from x

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
02-JAN-2005 01-JAN-2006
03-JAN-2005 01-JAN-2006
…
29-JAN-2005 01-JAN-2006
30-JAN-2005 01-JAN-2006
31-JAN-2005 01-JAN-2006
…
01-DEC-2005 01-JAN-2006
02-DEC-2005 01-JAN-2006
03-DEC-2005 01-JAN-2006
…
```

```
29-DEC-2005 01-JAN-2006
30-DEC-2005 01-JAN-2006
31-DEC-2005 01-JAN-2006
```

The final step is to use the function DAYNAME on the rows returned by the recursive view X and count how many times each weekday occurs. The final result is shown here:

```
with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (
             current_date -
         dayofyear(current_date) day)
         +1 day as start_date
  from t1
       ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
)
select dayname(start_date),count(*)
  from x
 group by dayname(start_date)

START_DATE   COUNT(*)
---------- ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

MySQL

This solution selects against table T500 to generate one row for every day in the year. The command on line 4 returns the first day of the current year. It does this by returning the year of the date returned by the function CURRENT_DATE and then appending a month and day (following MySQL's default date format). The result is shown here:

```
select concat(year(current_date),'-01-01')
  from t1

START_DATE
-----------
01-JAN-2005
```

Now that you have the first day in the current year, use the DATEADD function to add each value from T500.ID to generate each day in the year. Use the function DATE_FORMAT to return the weekday for each date. To generate the required number of rows from table T500, find the difference in days between the first day of the current year and the first day of the next year, and return that many rows (will be either 365 or 366). A portion of the results is shown here:

```
select date_format(
          date_add(
              cast(
            concat(year(current_date),'-01-01')
                   as date),
                   interval t500.id-1 day),
                   '%W') day
  from t500
 where t500.id <= datediff(
                        cast(
                     concat(year(current_date)+1,'-01-01')
                           as date),
                        cast(
                     concat(year(current_date),'-01-01')
                          as date))

DAY
-----------
01-JAN-2005
02-JAN-2005
03-JAN-2005
…
29-JAN-2005
30-JAN-2005
31-JAN-2005
…
01-DEC-2005
02-DEC-2005
03-DEC-2005
…
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

Now that you can return every day in the current year, count the occurrences of each weekday returned by the function DAYNAME. The final results are shown here:

```
select date_format(
          date_add(
              cast(
            concat(year(current_date),'-01-01')
                   as date),
                   interval t500.id-1 day),
                   '%W') day,
       count(*)
```

```
  from t500
 where t500.id <= datediff(
                        cast(
                      concat(year(current_date)+1,'-01-01')
                             as date),
                        cast(
                      concat(year(current_date),'-01-01')
                             as date))
 group by date_format(
             date_add(
                 cast(
               concat(year(current_date),'-01-01')
                      as date),
                     interval t500.id-1 day),
                     '%W')
```

```
DAY       COUNT(*)
--------- ----------
FRIDAY            52
MONDAY            52
SATURDAY          53
SUNDAY            52
THURSDAY          52
TUESDAY           52
WEDNESDAY         52
```

Oracle

The solutions provided either select against table T500 (a pivot table), or use the recursive CONNECT BY and WITH to generate a row for every day in the current year. The call to the function TRUNC truncates the current date to the first day of the current year.

If you are using the CONNECT BY/WITH solution, you can use the pseudo-column LEVEL to generate sequential numbers beginning at one. To generate the required number of rows needed for this solution, filter ROWNUM or LEVEL on the difference in days between the first day of the current year and the first day of the next year (will be 365 or 366 days). The next step is to increment each day by adding ROWNUM or LEVEL to the first day of the current year. Partial results are shown here:

```
/* Oracle 9i and later */
with x as (
select level lvl
  from dual
 connect by level <= (
   add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
 )
)
select trunc(sysdate,'y')+lvl-1   from x
```

If you are using the pivot-table solution, you can use any table or view with at least 366 rows in it. And since Oracle has ROWNUM, there's no need for a table with incrementing values starting from one. Consider the following example, which uses pivot table T500 to return every day in the current year:

```
/* Oracle 8i and earlier */
select trunc(sysdate,'y')+rownum-1 start_date
  from t500
 where rownum <= (add_months(trunc(sysdate,'y'),12)
                    - trunc(sysdate,'y'))

START_DATE
-----------
01-JAN-2005
02-JAN-2005
03-JAN-2005
…
29-JAN-2005
30-JAN-2005
31-JAN-2005
…
01-DEC-2005
02-DEC-2005
03-DEC-2005
…
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

Regardless of which approach you take, you eventually must use the function TO_CHAR to return the weekday name for each date and then count the occurrence of each name. The final results are shown here:

```
/* Oracle 9i and later */
with x as (
select level lvl
  from dual
 connect by level <= (
   add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
 )
)
select to_char(trunc(sysdate,'y')+lvl-1,'DAY'), count(*)
  from x
 group by to_char(trunc(sysdate,'y')+lvl-1,'DAY')

/* Oracle 8i and earlier */
select to_char(trunc(sysdate,'y')+rownum-1,'DAY') start_date,
       count(*)
  from t500
 where rownum <= (add_months(trunc(sysdate,'y'),12)
                    - trunc(sysdate,'y'))
 group by to_char(trunc(sysdate,'y')+rownum-1,'DAY')
```

```
START_DATE   COUNT(*)
---------- ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

PostgreSQL

The first step is to use the DATE_TRUNC function to return the year of the current date (shown here, selecting against T1 so only one row is returned):

```
select cast(
         date_trunc('year',current_date)
         as date) as start_date
  from t1

 START_DATE
 ----------
 01-JAN-2005
```

The next step is to select against a row source (any table expression, really) with at least 366 rows. The solution uses the function GENERATE_SERIES as the row source. You can, of course, use table T500 instead. Then add one day to the first day of the current year until you return every day in the year (shown here):

```
select cast( date_trunc('year',current_date)
               as date) + gs.id-1 as start_date
  from generate_series (1,366) gs(id)
 where gs.id <= (cast
                    ( date_trunc('year',current_date) +
                         interval '12 month' as date) -
     cast(date_trunc('year',current_date)
                      as date))

START_DATE
-----------
01-JAN-2005
02-JAN-2005
03-JAN-2005
…
29-JAN-2005
30-JAN-2005
31-JAN-2005
…
01-DEC-2005
02-DEC-2005
03-DEC-2005
```

```
…
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

The final step is to use the function TO_CHAR to return the weekday name for each date and then count the occurrence of each name. The final results are shown here:

```
select to_char(
          cast(
    date_trunc('year',current_date)
               as date) + gs.id-1,'DAY') as start_dates,
       count(*)
  from generate_series(1,366) gs(id)
 where gs.id <= (cast
                     ( date_trunc('year',current_date) +
                         interval '12 month' as date) -
     cast(date_trunc('year',current_date)
                      as date))
 group by to_char(
             cast(
       date_trunc('year',current_date)
          as date) + gs.id-1,'DAY')

START_DATE   COUNT(*)
---------- ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

SQL Server

Inline view TMP, in the recursive WITH view X, returns the first day of the current year and is shown here:

```
select cast(
       cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
  from t1

START_DATE
-----------
01-JAN-2005
```

Once you return the first day of the current year, add one year to START_DATE so that you have the beginning and end dates. You need to know both because you want to generate every day in a year.

START_DATE and END_DATE are shown here:

```
select start_date,
        dateadd(year,1,start_date) end_date
  from (
select cast(
       cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
  from t1
       ) tmp

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
```

Next, recursively increment START_DATE by one day and stop before it equals END_DATE. A portion of the rows returned by the recursive view X is shown below:

```
with x (start_date,end_date)
 as (
 select start_date,
        dateadd(year,1,start_date) end_date
   from (
 select cast(
        cast(year(getdate()) as varchar) + '-01-01'
             as datetime) start_date
   from t1
        ) tmp
 union all
 select dateadd(day,1,start_date), end_date
   from x
  where dateadd(day,1,start_date) < end_date
 )
 select * from x
 OPTION (MAXRECURSION 366)

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
02-JAN-2005 01-JAN-2006
03-JAN-2005 01-JAN-2006
…
29-JAN-2005 01-JAN-2006
30-JAN-2005 01-JAN-2006
31-JAN-2005 01-JAN-2006
…
01-DEC-2005 01-JAN-2006
02-DEC-2005 01-JAN-2006
03-DEC-2005 01-JAN-2006
…
29-DEC-2005 01-JAN-2006
30-DEC-2005 01-JAN-2006
31-DEC-2005 01-JAN-2006
```

The final step is to use the function DATENAME on the rows returned by the recursive view X and count how many times each weekday occurs. The final result is shown here:

```
with x(start_date,end_date)
 as (
 select start_date,
        dateadd(year,1,start_date) end_date
   from (
 select cast(
        cast(year(getdate()) as varchar) + '-01-01'
             as datetime) start_date
   from t1
        ) tmp
 union all
 select dateadd(day,1,start_date), end_date
   from x
  where dateadd(day,1,start_date) < end_date
 )
 select datename(dw,start_date), count(*)
   from x
  group by datename(dw,start_date)
 OPTION (MAXRECURSION 366)


START_DATE   COUNT(*)
---------  ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

8.7 Determining the Date Difference Between the Current Record and the Next Record

Problem

You want to determine the difference in days between two dates (specifically dates stored in two different rows). For example, for every employee in DEPTNO 10, you want to determine the number of days between the day they were hired and the day the next employee (can be in another department) was hired.

Solution

The trick to this problem's solution is to find the earliest HIREDATE after the current employee was hired. After that, simply use the technique from Recipe 8.2 to find the difference in days.

DB2

Use a scalar subquery to find the next HIREDATE relative to the current HIREDATE. Then use the DAYS function to find the difference in days:

```
select x.*,
       days(x.next_hd) - days(x.hiredate) diff
  from (
select e.deptno, e.ename, e.hiredate,
       lead(hiredate)over(order by hiredate) next_hd
  from emp e
 where e.deptno = 10
       ) x
```

MySQL and SQL Server

Use the lead function to access the next row. The SQL Server version of DATEDIFF is used here:

```
select x.ename, x.hiredate, x.next_hd,
       datediff(x.hiredate,x.next_hd,day) as diff
  from (
select deptno, ename, hiredate,
       lead(hiredate)over(order by hiredate) as next_hd
  from emp e
       ) x
 where e.deptno=10
```

MySQL users can exclude the first argument ("day") and switch the order of the two remaining arguments:

```
        datediff(x.next_hd, x.hiredate) diff
```

Oracle

Use the window function LEAD OVER to access the next HIREDATE relative to the current row, thus facilitating subtraction:

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate)over(order by hiredate) next_hd
  from emp
       )
 where deptno=10
```

PostgreSQL

Use a scalar subquery to find the next HIREDATE relative to the current HIREDATE. Then use simple subtraction to find the difference in days:

```
1 select x.*,
2        x.next_hd - x.hiredate as diff
3   from (
4 select e.deptno, e.ename, e.hiredate,
5        lead(hiredate)over(order by hiredate) as next_hd
7   from emp e
8  where e.deptno = 10
9        ) x
```

Discussion

Despite the differences in syntax, the approach is the same for all these solutions: use the window function LEAD and then find the difference in days between the two using the technique described in Recipe 8.2.

The ability to access rows around your current row without additional joins provides for more readable and efficient code. When working with window functions, keep in mind that they are evaluated after the WHERE clause, hence the need for an inline view in the solution. If you were to move the filter on DEPTNO into the inline view, the results would change (only the HIREDATEs from DEPTNO 10 would be considered). One important note to mention about Oracle's LEAD and LAG functions is their behavior in the presence of duplicates. In the preface we mention that these recipes are not coded "defensively" because there are too many conditions that one can't possibly foresee that can break code. Or, even if one can foresee every problem, sometimes the resulting SQL becomes unreadable. So in most cases, the goal of a solution is to introduce a technique: one that you can use in your production system, but that must be tested and many times tweaked to work for your particular data. In this case, though, there is a situation that we will discuss simply because the workaround may not be all that obvious, particularly for those coming from non-Oracle systems. In this example there are no duplicate HIREDATEs in table EMP, but it is certainly possible (and probably likely) that there are duplicate date values in your tables. Consider the employees in DEPTNO 10 and their HIREDATEs:

```
select ename, hiredate
  from emp
 where deptno=10
 order by 2

ENAME  HIREDATE
------ -----------
CLARK  09-JUN-2006
KING   17-NOV-2006
MILLER 23-JAN-2007
```

For the sake of this example, let's insert four duplicates such that there are five employees (including KING) hired on November 17:

```
insert into emp (empno,ename,deptno,hiredate)
values (1,'ant',10,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,hiredate)
values (2,'joe',10,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,hiredate)
values (3,'jim',10,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,hiredate)
values (4,'choi',10,to_date('17-NOV-2006'))

select ename, hiredate
  from emp
 where deptno=10
 order by 2

ENAME  HIREDATE
------ -----------
CLARK  09-JUN-2006
ant    17-NOV-2006
joe    17-NOV-2006
KING   17-NOV-2006
jim    17-NOV-2006
choi   17-NOV-2007
MILLER 23-JAN-2007
```

Now there are multiple employees in DEPTNO 10 hired on the same day. If you try to use the proposed solution (moving the filter into the inline view so you only are concerned with employees in DEPTNO 10 and their HIREDATEs) on this result set, you get the following output:

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate)over(order by hiredate) next_hd
  from emp
 where deptno=10
        )

ENAME  HIREDATE    NEXT_HD           DIFF
------ ----------- ----------- ----------
CLARK  09-JUN-2006 17-NOV-2006        161
ant    17-NOV-2006 17-NOV-2006          0
joe    17-NOV-2006 17-NOV-2006          0
KING   17-NOV-2006 17-NOV-2006          0
jim    17-NOV-2006 17-NOV-2006          0
```

```
choi   17-NOV-2006 23-JAN-2007         67
MILLER 23-JAN-2007 (null)          (null)
```

Looking at the values of DIFF for four of the five employees hired on the same day, you can see that the value is zero. This is not correct. All employees hired on the same day should have their dates evaluated against the HIREDATE of the next date on which an employee was hired (i.e., all employees hired on November 17 should be evaluated against MILLER's HIREDATE). The problem here is that the LEAD function orders the rows by HIREDATE but does not skip duplicates. So, for example, when employee ANT's HIREDATE is evaluated against employee JOE's HIREDATE, the difference is zero, hence a DIFF value of zero for ANT. Fortunately, Oracle has provided an easy workaround for situations like this one. When invoking the LEAD function, you can pass an argument to LEAD to specify exactly where the future row is (i.e., is it the next row, 10 rows later, etc.). So, looking at employee ANT, instead of looking ahead one row, you need to look ahead five rows (you want to jump over all the other duplicates), because that's where MILLER is. If you look at employee JOE, he is four rows from MILLER, JIM is three rows from MILLER, KING is two rows from MILLER, and pretty boy CHOI is one row from MILLER. To get the correct answer, simply pass the distance from each employee to MILLER as an argument to LEAD. The solution is shown here:

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate,cnt-rn+1)over(order by hiredate) next_hd
  from (
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
        )
        )

ENAME  HIREDATE    NEXT_HD           DIFF
------ ----------- ----------- ----------
CLARK  09-JUN-2006 17-NOV-2006        161
ant    17-NOV-2006 23-JAN-2007         67
joe    17-NOV-2006 23-JAN-2007         67
jim    17-NOV-2006 23-JAN-2007         67
choi   17-NOV-2006 23-JAN-2007         67
KING   17-NOV-2006 23-JAN-2007         67
MILLER 23-JAN-2007 (null)          (null)
```

Now the results are correct. All the employees hired on the same day have their HIREDATEs evaluated against the next HIREDATE, not a HIREDATE that matches their own. If the workaround isn't immediately obvious, simply break down the query.

Start with the inline view:

```
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10

DEPTNO ENAME  HIREDATE           CNT         RN
------ ------ ----------- ---------- ----------
    10 CLARK  09-JUN-2006          1          1
    10 ant    17-NOV-2006          5          1
    10 joe    17-NOV-2006          5          2
    10 jim    17-NOV-2006          5          3
    10 choi   17-NOV-2006          5          4
    10 KING   17-NOV-2006          5          5
    10 MILLER 23-JAN-2007          1          1
```

The window function COUNT OVER counts the number of times each HIREDATE occurs and returns this value to each row. For the duplicate HIREDATEs, a value of 5 is returned for each row with that HIREDATE. The window function ROW_NUMBER OVER ranks each employee by EMPNO. The ranking is partitioned by HIREDATE, so unless there are duplicate HIREDATEs, each employee will have a rank of 1. At this point, all the duplicates have been counted and ranked, and the ranking can serve as the distance to the next HIREDATE (MILLER's HIREDATE). You can see this by subtracting RN from CNT and adding 1 for each row when calling LEAD:

```
select deptno, ename, hiredate,
       cnt-rn+1 distance_to_miller,
       lead(hiredate,cnt-rn+1)over(order by hiredate) next_hd
  from (
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
        )

DEPTNO ENAME  HIREDATE    DISTANCE_TO_MILLER NEXT_HD
------ ------ ----------- ------------------ -----------
    10 CLARK  09-JUN-2006                  1 17-NOV-2006
    10 ant    17-NOV-2006                  5 23-JAN-2007
    10 joe    17-NOV-2006                  4 23-JAN-2007
    10 jim    17-NOV-2006                  3 23-JAN-2007
    10 choi   17-NOV-2006                  2 23-JAN-2007
    10 KING   17-NOV-2006                  1 23-JAN-2007
    10 MILLER 23-JAN-2007                  1 (null)
```

As you can see, by passing the appropriate distance to jump ahead to, the LEAD function performs the subtraction on the correct dates.

8.8 Summing Up

Dates are a common data type, but have their own quirks, as they have more structure than simple number data types. In relative terms, there is less standardization between vendors than in many other areas, but every implementation has a core group of functions that perform the same tasks even where the syntax is slightly different. Mastering this core group will ensure your success with dates.

CHAPTER 9

Date Manipulation

This chapter introduces recipes for searching and modifying dates. Queries involving dates are very common. Thus, you need to know how to think when working with dates, and you need to have a good understanding of the functions that your RDBMS platform provides for manipulating them. The recipes in this chapter form an important foundation for future work as you move on to more complex queries involving not only dates, but times, too.

Before getting into the recipes, we want to reinforce the concept (mentioned in the preface) of using these solutions as guidelines to solving your specific problems. Try to think "big picture." For example, if a recipe solves a problem for the current month, keep in mind that you may be able to use the recipe for any month (with minor modifications), not just the month used in the recipe. Again, these recipes are guidelines, the absolute final option. There's no possible way a book can contain an answer for all your problems, but if you understand what is presented here, modifying these solutions to fit your needs is trivial. Also consider alternative versions of these solutions. For instance, if the solution uses one particular function provided by your RDBMS, it is worth the time and effort to find out if there is an alternative—maybe one that is more or less efficient than what is presented here. Knowing your options will make you a better SQL programmer.

The recipes presented in this chapter use simple date data types. If you are using more complex date data types, you will need to adjust the solutions accordingly.

9.1 Determining Whether a Year Is a Leap Year

Problem

You want to determine whether the current year is a leap year.

Solution

If you've worked on SQL for some time, there's no doubt that you've come across several techniques for solving this problem. Just about all the solutions we've encountered work well, but the one presented in this recipe is probably the simplest. This solution simply checks the last day of February; if it is the 29th, then the current year is a leap year.

DB2

Use the recursive WITH clause to return each day in February. Use the aggregate function MAX to determine the last day in February:

```
 1   with x (dy,mth)
 2     as (
 3 select dy, month(dy)
 4   from (
 5 select (current_date -
 6          dayofyear(current_date) days +1 days)
 7           +1 months as dy
 8   from t1
 9        ) tmp1
10  union all
11 select dy+1 days, mth
12   from x
13  where month(dy+1 day) = mth
14 )
15 select max(day(dy))
16   from x
```

Oracle

Use the function LAST_DAY to find the last day in February:

```
1 select to_char(
2          last_day(add_months(trunc(sysdate,'y'),1)),
3         'DD')
4   from t1
```

PostgreSQL

Use the function GENERATE_SERIES to return each day in February, and then use the aggregate function MAX to find the last day in February:

```
select max(to_char(tmp2.dy+x.id,'DD')) as dy
  from (
select dy, to_char(dy,'MM') as mth
  from (
select cast(cast(
            date_trunc('year',current_date) as date)
                       + interval '1 month' as date) as dy
  from t1
       ) tmp1
       ) tmp2, generate_series (0,29) x(id)
 where to_char(tmp2.dy+x.id,'MM') = tmp2.mth
```

MySQL

Use the function LAST_DAY to find the last day in February:

```
select day(
       last_day(
       date_add(
       date_add(
       date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day),
                interval 1 month))) dy
  from t1
```

SQL Server

Use the recursive WITH clause to return each day in February. Use the aggregate function MAX to determine the last day in February:

```
select coalesce
               (day
                       (cast(concat
                       (year(getdate()),'-02-29')
                        as date))
                        ,28);
```

Discussion

DB2

The inline view TMP1 in the recursive view X returns the first day in February by:

1. Starting with the current date

2. Using DAYOFYEAR to determine the number of days into the current year that the current date represents
3. Subtracting that number of days from the current date to get December 31 of the prior year and then adding one to get to January 1 of the current year
4. Adding one month to get to February 1

The result of all this math is shown here:

```
 select (current_date
          dayofyear(current_date) days +1 days) +1 months as dy
   from t1

DY
-----------
01-FEB-2005
```

The next step is to return the month of the date returned by inline view TMP1 by using the MONTH function:

```
select dy, month(dy) as mth
  from (
select (current_date
          dayofyear(current_date) days +1 days) +1 months as dy
  from t1
       ) tmp1

DY          MTH
----------- ---
01-FEB-2005   2
```

The results presented thus far provide the start point for the recursive operation that generates each day in February. To return each day in February, repeatedly add one day to DY until you are no longer in the month of February. A portion of the results of the WITH operation is shown here:

```
  with x (dy,mth)
    as (
select dy, month(dy)
  from (
select (current_date -
         dayofyear(current_date) days +1 days) +1 months as dy
  from t1
       ) tmp1
 union all
 select dy+1 days, mth
   from x
  where month(dy+1 day) = mth
 )
 select dy,mth
   from x
```

```
DY          MTH
----------- ---
01-FEB-2005   2
…
10-FEB-2005   2
…
28-FEB-2005   2
```

The final step is to use the MAX function on the DY column to return the last day in February; if it is the 29th, you are in a leap year.

Oracle

The first step is to find the beginning of the year using the TRUNC function:

```
select trunc(sysdate,'y')
  from t1

DY
-----------
01-JAN-2005
```

Because the first day of the year is January 1st, the next step is to add one month to get to February 1st:

```
select add_months(trunc(sysdate,'y'),1) dy
  from t1

DY
-----------
01-FEB-2005
```

The next step is to use the LAST_DAY function to find the last day in February:

```
select last_day(add_months(trunc(sysdate,'y'),1)) dy
  from t1

DY
-----------
28-FEB-2005
```

The final step (which is optional) is to use TO_CHAR to return either 28 or 29.

PostgreSQL

The first step is to examine the results returned by inline view TMP1. Use the DATE_TRUNC function to find the beginning of the current year and cast that result as a DATE:

```
select cast(date_trunc('year',current_date) as date) as dy
  from t1
```

```
DY
-----------
01-JAN-2005
```

The next step is to add one month to the first day of the current year to get the first day in February, casting the result as a date:

```
select cast(cast(
            date_trunc('year',current_date) as date)
                       + interval '1 month' as date) as dy
  from t1

DY
-----------
01-FEB-2005
```

Next, return DY from inline view TMP1 along with the numeric month of DY. Return the numeric month by using the TO_CHAR function:

```
select dy, to_char(dy,'MM') as mth
   from (
 select cast(cast(
             date_trunc('year',current_date) as date)
                        + interval '1 month' as date) as dy
   from t1
        ) tmp1

DY          MTH
----------- ---
01-FEB-2005   2
```

The results shown thus far comprise the result set of inline view TMP2. Your next step is to use the extremely useful function GENERATE_SERIES to return 29 rows (values 1 through 29). Every row returned by GENERATE_SERIES (aliased X) is added to DY from inline view TMP2. Partial results are shown here:

```
select tmp2.dy+x.id as dy, tmp2.mth
  from (
select dy, to_char(dy,'MM') as mth
  from (
select cast(cast(
            date_trunc('year',current_date) as date)
                       + interval '1 month' as date) as dy
  from t1
       ) tmp1
       ) tmp2, generate_series (0,29) x(id)
 where to_char(tmp2.dy+x.id,'MM') = tmp2.mth

DY          MTH
----------- ---
01-FEB-2005  02
…
10-FEB-2005  02
```

```
…
28-FEB-2005  02
```

The final step is to use the MAX function to return the last day in February. The function TO_CHAR is applied to that value and will return either 28 or 29.

MySQL

The first step is to find the first day of the current year by subtracting from the current date the number of days it is into the year and then adding one day. Do all of this with the DATE_ADD function:

```
select date_add(
       date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day) dy
  from t1

DY
-----------
01-JAN-2005
```

Then add one month again using the DATE_ADD function:

```
select date_add(
       date_add(
       date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day),
                interval 1 month) dy
  from t1

DY
-----------
01-FEB-2005
```

Now that you've made it to February, use the LAST_DAY function to find the last day of the month:

```
select last_day(
       date_add(
       date_add(
       date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day),
                interval 1 month)) dy
  from t1

DY
-----------
28-FEB-2005
```

The final step (which is optional) is to use the DAY function to return either a 28 or 29.

SQL Server

We can create a new date in most RDMSs by creating a string in a recognized date format and using CAST to change format. We can therefore use the current year by retrieving the year from the current date. In SQL Server, this is done by applying YEAR to GET_DATE:

```
select YEAR(GETDATE());
```

This will return the year as an integer. We can then create 29th of February by using CONCAT and CAST:

```
select cast(concat
                                    (year(getdate()),'-02-29');
```

However, this won't be a *real* date if the current year isn't a leap year. For example, there is no date 2019-02-29. Hence, if we try to use an operator like DAY to find any of its parts, it will return NULL. Therefore, use COALESCE and DAY to determine whether there is a 29th day in the month.

9.2 Determining the Number of Days in a Year

Problem

You want to count the number of days in the current year.

Solution

The number of days in the current year is the difference between the first day of the next year and the first day of the current year (in days). For each solution the steps are:

1. Find the first day of the current year.
2. Add one year to that date (to get the first day of the next year).
3. Subtract the current year from the result of Step 2.

The solutions differ only in the built-in functions that you use to perform these steps.

DB2

Use the function DAYOFYEAR to help find the first day of the current year, and use DAYS to find the number of days in the current year:

```
1 select days((curr_year + 1 year)) - days(curr_year)
2   from (
3 select (current_date -
4        dayofyear(current_date) day +
5         1 day) curr_year
6   from t1
7       ) x
```

Oracle

Use the function TRUNC to find the beginning of the current year, and use ADD_MONTHS to then find the beginning of next year:

```
1 selectadd_months(trunc(sysdate,'y'),12) - trunc(sysdate,'y')
2   from dual
```

PostgreSQL

Use the function DATE_TRUNC to find the beginning of the current year. Then use interval arithmetic to determine the beginning of next year:

```
1 select cast((curr_year + interval '1 year') as date) - curr_year
2   from (
3 select cast(date_trunc('year',current_date) as date) as curr_year
4   from t1
5       ) x
```

MySQL

Use ADDDATE to help find the beginning of the current year. Use DATEDIFF and interval arithmetic to determine the number of days in the year:

```
1 select datediff((curr_year + interval 1 year),curr_year)
2   from (
3 select adddate(current_date,-dayofyear(current_date)+1) curr_year
4   from t1
5       ) x
```

SQL Server

Use the function DATEADD to find the first day of the current year. Use DATEDIFF to return the number of days in the current year:

```
1 select datediff(d,curr_year,dateadd(yy,1,curr_year))
2   from (
3 select dateadd(d,-datepart(dy,getdate())+1,getdate()) curr_year
4   from t1
5       ) x
```

Discussion

DB2

The first step is to find the first day of the current year. Use DAYOFYEAR to determine how many days you are into the current year. Subtract that value from the current date to get the last day of last year, and then add 1:

```
select (current_date
        dayofyear(current_date) day +
         1 day) curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

Now that you have the first day of the current year, just add one year to it; this gives you the first day of next year. Then subtract the beginning of the current year from the beginning of the next year.

Oracle

The first step is to find the first day of the current year, which you can easily do by invoking the built-in TRUNC function and passing Y as the second argument (thereby truncating the date to the beginning of the year):

```
select select trunc(sysdate,'y') curr_year
  from dual

CURR_YEAR
-----------
01-JAN-2005
```

Then add one year to arrive at the first day of the next year. Finally, subtract the two dates to find the number of days in the current year.

PostgreSQL

Begin by finding the first day of the current year. To do that, invoke the DATE_TRUNC function as follows:

```
select cast(date_trunc('year',current_date) as date) as curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

You can then easily add a year to compute the first day of next year. Then all you need to do is to subtract the two dates. Be sure to subtract the earlier date from the later date. The result will be the number of days in the current year.

MySQL

Your first step is to find the first day of the current year. Use DAYOFYEAR to find how many days you are into the current year. Subtract that value from the current date, and add one:

```
select adddate(current_date,-dayofyear(current_date)+1) curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

Now that you have the first day of the current year, your next step is to add one year to it to get the first day of next year. Then subtract the beginning of the current year from the beginning of the next year. The result is the number of days in the current year.

SQL Server

Your first step is to find the first day of the current year. Use DATEADD and DATEPART to subtract from the current date the number of days into the year the current date is, and add 1:

```
select dateadd(d,-datepart(dy,getdate())+1,getdate()) curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

Now that you have the first day of the current year, your next step is to add one year to it to get the first day of the next year. Then subtract the beginning of the current year from the beginning of the next year. The result is the number of days in the current year.

9.3 Extracting Units of Time from a Date

Problem

You want to break the current date down into six parts: day, month, year, second, minute, and hour. You want the results to be returned as numbers.

Solution

Use of the current date is arbitrary. Feel free to use this recipe with other dates. Most vendors have now adopted the ANSI standard function for extracting parts of dates, EXTRACT, although SQL Server is an exception. They also retain their own legacy methods.

DB2

DB2 implements a set of built-in functions that make it easy for you to extract portions of a date. The function names HOUR, MINUTE, SECOND, DAY, MONTH, and YEAR conveniently correspond to the units of time you can return: if you want the day, use DAY; hour, use HOUR; etc. For example:

```
1 select    hour( current_timestamp ) hr,
2         minute( current_timestamp ) min,
3         second( current_timestamp ) sec,
4            day( current_timestamp ) dy,
5          month( current_timestamp ) mth,
6            year( current_timestamp ) yr
7   from t1
```

```
select
        extract(hour from current_timestamp)
      , extract(minute from current_timestamp
      , extract(second from current_timestamp)
      , extract(day from current_timestamp)
      , extract(month from current_timestamp)
      , extract(year from current_timestamp)

          HR   MIN   SEC    DY   MTH    YR
        ---- ----- ----- ----- ----- -----
          20    28    36    15     6  2005
```

Oracle

Use functions TO_CHAR and TO_NUMBER to return specific units of time from a date:

```
1  select to_number(to_char(sysdate,'hh24')) hour,
2         to_number(to_char(sysdate,'mi')) min,
3         to_number(to_char(sysdate,'ss')) sec,
4         to_number(to_char(sysdate,'dd')) day,
5         to_number(to_char(sysdate,'mm')) mth,
6         to_number(to_char(sysdate,'yyyy')) year
7   from dual

  HOUR   MIN   SEC   DAY   MTH  YEAR
  ---- ----- ----- ----- ----- -----
    20    28    36    15     6  2005
```

PostgreSQL

Use functions TO_CHAR and TO_NUMBER to return specific units of time from a date:

```
1 select to_number(to_char(current_timestamp,'hh24'),'99') as hr,
2        to_number(to_char(current_timestamp,'mi'),'99') as min,
3        to_number(to_char(current_timestamp,'ss'),'99') as sec,
4        to_number(to_char(current_timestamp,'dd'),'99') as day,
5        to_number(to_char(current_timestamp,'mm'),'99') as mth,
6        to_number(to_char(current_timestamp,'yyyy'),'9999') as yr
7   from t1

   HR   MIN   SEC   DAY   MTH    YR
 ---- ----- ----- ----- ----- -----
 20      28    36    15     6  2005
```

MySQL

Use the DATE_FORMAT function to return specific units of time from a date:

```
1 select date_format(current_timestamp,'%k') hr,
2        date_format(current_timestamp,'%i') min,
3        date_format(current_timestamp,'%s') sec,
4        date_format(current_timestamp,'%d') dy,
5        date_format(current_timestamp,'%m') mon,
6        date_format(current_timestamp,'%Y') yr
7   from t1

  HR   MIN   SEC   DAY   MTH    YR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

SQL Server

Use the function DATEPART to return specific units of time from a date:

```
1 select datepart( hour, getdate()) hr,
2        datepart( minute,getdate()) min,
3        datepart( second,getdate()) sec,
4        datepart( day, getdate()) dy,
5        datepart( month, getdate()) mon,
6        datepart( year, getdate()) yr
7   from t1

  HR   MIN   SEC   DAY   MTH    YR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

Discussion

There's nothing fancy in these solutions; just take advantage of what you're already paying for. Take the time to learn the date functions available to you. This recipe only scratches the surface of the functions presented in each solution. You'll find that each of the functions takes many more arguments and can return more information than what this recipe provides you.

9.4 Determining the First and Last Days of a Month

Problem

You want to determine the first and last days for the current month.

Solution

The solutions presented here are for finding first and last days for the current month. Using the current month is arbitrary. With a bit of adjustment, you can make the solutions work for any month.

DB2

Use the DAY function to return the number of days into the current month the current date represents. Subtract this value from the current date, and then add one to get the first of the month. To get the last day of the month, add one month to the current date, and then subtract from it the value returned by the DAY function as applied to the current date:

```
1 select (date(current_date) - day(date(current_date)) day + 1 day) firstday,
2        (date(current_date)+1 month
3         - day(date(current_date)+1 month) day) lastday
4   from t1
```

Oracle

Use the function TRUNC to find the first of the month, and use the function LAST_DAY to find the last day of the month:

```
1 select trunc(sysdate,'mm') firstday,
2        last_day(sysdate) lastday
3   from dual
```

Using TRUNC as described here will result in the loss of any time-of-day component, whereas LAST_DAY will preserve the time of day.

PostgreSQL

Use the DATE_TRUNC function to truncate the current date to the first of the current month. Once you have the first day of the month, add one month and subtract one day to find the end of the current month:

```
select firstday,
       cast(firstday + interval '1 month'
                     - interval '1 day' as date) as lastday
  from (
select cast(date_trunc('month',current_date) as date) as firstday
  from t1
       ) x
```

MySQL

Use the DATE_ADD and DAY functions to find the number of days into the month the current date is. Then subtract that value from the current date and add one to find the first of the month. To find the last day of the current month, use the LAST_DAY function:

```
select date_add(current_date,
                interval -day(current_date)+1 day) firstday,
       last_day(current_date) lastday
  from t1
```

SQL Server

Use the DATEADD and DAY functions to find the number of days into the month represented by the current date. Then subtract that value from the current date and add one to find the first of the month. To get the last day of the month, add one month to the current date, and then subtract from that result the value returned by the DAY function applied to the current date, again using the functions DAY and DATEADD:

```
select dateadd(day,-day(getdate())+1,getdate()) firstday,
       dateadd(day,
               -day(dateadd(month,1,getdate())),
               dateadd(month,1,getdate())) lastday
  from t1
```

Discussion

DB2

To find the first day of the month, simply find the numeric value of the current day of the month, and then subtract this from the current date. For example, if the date is March 14th, the numeric day value is 14. Subtracting 14 days from March 14th gives you the last day of the month in February. From there, simply add one day to get to

the first of the current month. The technique to get the last day of the month is similar to that of the first: subtract the numeric day of the month from the current date to get the last day of the prior month. Since we want the last day of the current month (not the last day of the prior month), we need to add one month to the current date.

Oracle

To find the first day of the current month, use the TRUNC function with "mm" as the second argument to "truncate" the current date down to the first of the month. To find the last day of the current month, simply use the LAST_DAY function.

PostgreSQL

To find the first day of the current month, use the DATE_TRUNC function with "month" as the second argument to "truncate" the current date down to the first of the month. To find the last day of the current month, add one month to the first day of the month, and then subtract one day.

MySQL

To find the first day of the month, use the DAY function. The DAY function returns the day of the month for the date passed. If you subtract the value returned by DAY(CURRENT_DATE) from the current date, you get the last day of the prior month; add one day to get the first day of the current month. To find the last day of the current month, simply use the LAST_DAY function.

SQL Server

To find the first day of the month, use the DAY function. The DAY function conveniently returns the day of the month for the date passed. If you subtract the value returned by DAY(GETDATE()) from the current date, you get the last day of the prior month; add one day to get the first day of the current month. To find the last day of the current month, use the DATEADD function. Add one month to the current date, then subtract from it the value returned by DAY(GETDATE()) to get the last day of the current month. Add one month to the current date, and then subtract from it the value returned by DAY(DATEADD(MONTH,1,GETDATE())) to get the last day of the current month.

9.5 Determining All Dates for a Particular Weekday Throughout a Year

Problem

You want to find all the dates in a year that correspond to a given day of the week. For example, you may want to generate a list of Fridays for the current year.

Solution

Regardless of vendor, the key to the solution is to return each day for the current year and keep only those dates corresponding to the day of the week that you care about. The solution examples retain all the Fridays.

DB2

Use the recursive WITH clause to return each day in the current year. Then use the function DAYNAME to keep only Fridays:

```
with x (dy,yr)
  as (
select dy, year(dy) yr
  from (
select (current_date -
        dayofyear(current_date) days +1 days) as dy
  from t1
      ) tmp1
 union all
select dy+1 days, yr
  from x
 where year(dy +1 day) = yr
)
select dy
  from x
 where dayname(dy) = 'Friday'
```

Oracle

Use the recursive CONNECT BY clause to return each day in the current year. Then use the function TO_CHAR to keep only Fridays:

```
with x
  as (
select trunc(sysdate,'y')+level-1 dy
  from t1
  connect by level <=
     add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
)
select *
```

```
from x
where to_char( dy, 'dy') = 'fri'
```

PostgreSQL

Use a recursive CTE to generate every day of the year, and filter out days that aren't Fridays. This version makes use of the ANSI standard EXTRACT, so it will run on a wide variety of RDBMs:

```
with recursive cal (dy)
as (
select current_date
 -(cast
  (extract(doy from current_date) as integer)
 -1)
 union all
 select dy+1
 from cal
 where extract(year from dy)=extract(year from (dy+1))
   )

 select dy,extract(dow from dy) from cal
 where cast(extract(dow from dy) as integer) = 6
```

MySQL

Use a recursive CTE to find all the days in the year. Then filter all days but Fridays:

```
        with recursive cal (dy,yr)
as
  (
  select dy, extract(year from dy) as yr
from
  (select adddate
          (adddate(current_date, interval - dayofyear(current_date)
day), interval 1 day) as dy) as tmp1
union all
   select date_add(dy, interval 1 day), yr
from cal
where extract(year from date_add(dy, interval 1 day)) = yr
)
   select dy from cal
   where dayofweek(dy) = 6
```

SQL Server

Use the recursive WITH clause to return each day in the current year. Then use the function DAYNAME to keep only Fridays:

```
 with x (dy,yr)
   as (
select dy, year(dy) yr
```

```
 4   from (
 5 select getdate()-datepart(dy,getdate())+1 dy
 6   from t1
 7        ) tmp1
 8  union all
 9 select dateadd(dd,1,dy), yr
10   from x
11  where year(dateadd(dd,1,dy)) = yr
12 )
13 select x.dy
14   from x
15  where datename(dw,x.dy) = 'Friday'
16 option (maxrecursion 400)
```

Discussion

DB2

To find all the Fridays in the current year, you must be able to return every day in the current year. The first step is to find the first day of the year by using the DAYOFYEAR function. Subtract the value returned by DAYOFYEAR(CURRENT_DATE) from the current date to get December 31 of the prior year, and then add one to get the first day of the current year:

```
select (current_date
        dayofyear(current_date) days +1 days) as dy
  from t1

DY
-----------
01-JAN-2005
```

Now that you have the first day of the year, use the WITH clause to repeatedly add one day to the first day of the year until you are no longer in the current year. The result set will be every day in the current year (a portion of the rows returned by the recursive view X is shown here):

```
 with x (dy,yr)
   as (
select dy, year(dy) yr
  from (
select (current_date
        dayofyear(current_date) days +1 days) as dy
  from t1
       ) tmp1
union all
select dy+1 days, yr
  from x
 where year(dy +1 day) = yr
)
select dy
```

```
  from x

DY
-----------
01-JAN-2020
…
15-FEB-2020
…
22-NOV-2020
…
31-DEC-2020
```

The final step is to use the DAYNAME function to keep only rows that are Fridays.

Oracle

To find all the Fridays in the current year, you must be able to return every day in the current year. Begin by using the TRUNC function to find the first day of the year:

```
select trunc(sysdate,'y') dy
          from t1

        DY
        -----------
        01-JAN-2020
```

Next, use the CONNECT BY clause to return every day in the current year (to understand how to use CONNECT BY to generate rows, see Recipe 10.5).

As an aside, this recipe uses the WITH clause, but you can also use an inline view.

A portion of the result set returned by view X is shown here:

```
 with x
   as (
select trunc(sysdate,'y')+level-1 dy
from t1
 connect by level <=
    add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
)
select *
from x

DY
-----------
01-JAN-2020
…
15-FEB-2020
```

```
…
22-NOV-2020
…
31-DEC-2020
```

The final step is to use the TO_CHAR function to keep only Fridays.

PostgreSQL

To find the Fridays, first find all the days. You need to find the first day of the year, and then use the recursive CTE to fill in the rest of the days. Remember PostgreSQL is one of the packages that requires the use of the RECURSIVE keyword to identify a recursive CTE.

The final step is to use the TO_CHAR function to keep only the Fridays.

MySQL

To find all the Fridays in the current year, you must be able to return every day in the current year. The first step is to find the first day of the year. Subtract the value returned by DAYOFYEAR(CURRENT_DATE) from the current date, and then add one to get the first day of the current year:

```
select adddate(
       adddate(current_date,
               interval -dayofyear(current_date) day),
               interval 1 day ) dy
  from t1

DY
-----------
01-JAN-2020
```

Once you've got the first day of the year, it's simple to use a recursive CTE to add every day of the year:

```
with cal (dy) as
(select current

union all
select dy+1


        DY
        -----------
        01-JAN-2020
        …
        15-FEB-2020
        …
        22-NOV-2020
        …
        31-DEC-2020
```

The final step is to use the DAYNAME function to keep only Fridays.

SQL Server

To find all the Fridays in the current year, you must be able to return every day in the current year. The first step is to find the first day of the year by using the DATEPART function. Subtract the value returned by DATEPART(DY,GETDATE()) from the current date, and then add one to get the first day of the current year:

```
select getdate()-datepart(dy,getdate())+1 dy
  from t1

DY
-----------
01-JAN-2005
```

Now that you have the first day of the year, use the WITH clause and the DATEADD function to repeatedly add one day to the first day of the year until you are no longer in the current year. The result set will be every day in the current year (a portion of the rows returned by the recursive view X is shown here):

```
with x (dy,yr)
  as (
select dy, year(dy) yr
  from (
select getdate()-datepart(dy,getdate())+1 dy
  from t1
       ) tmp1
 union all
select dateadd(dd,1,dy), yr
  from x
 where year(dateadd(dd,1,dy)) = yr
)
select x.dy
  from x
option (maxrecursion 400)

DY
-----------
01-JAN-2020
…
15-FEB-2020
…
22-NOV-2020
…
31-DEC-2020
```

Finally, use the DATENAME function to keep only rows that are Fridays. For this solution to work, you must set MAXRECURSION to at least 366 (the filter on the year portion of the current year, in recursive view X, guarantees you will never generate more than 366 rows).

9.6 Determining the Date of the First and Last Occurrences of a Specific Weekday in a Month

Problem

You want to find, for example, the first and last Mondays of the current month.

Solution

The choice to use Monday and the current month is arbitrary; you can use the solutions presented in this recipe for any weekday and any month. Because each weekday is 7 days apart from itself, once you have the first instance of a weekday, you can add 7 days to get the second and 14 days to get the third. Likewise, if you have the last instance of a weekday in a month, you can subtract 7 days to get the third and subtract 14 days to get the second.

DB2

Use the recursive WITH clause to generate each day in the current month and use a CASE expression to flag all Mondays. The first and last Mondays will be the earliest and latest of the flagged dates:

```
 1   with x (dy,mth,is_monday)
 2     as (
 3 select dy,month(dy),
 4        case when dayname(dy)='Monday'
 5              then 1 else 0
 6        end
 7   from (
 8 select (current_date-day(current_date) day +1 day) dy
 9   from t1
10        ) tmp1
11  union all
12 select (dy +1 day), mth,
13        case when dayname(dy +1 day)='Monday'
14             then 1 else 0
15        end
16   from x
17  where month(dy +1 day) = mth
18 )
19 select min(dy) first_monday, max(dy) last_monday
20   from x
21  where is_monday = 1
```

Oracle

Use the functions NEXT_DAY and LAST_DAY, together with a bit of clever date arithmetic, to find the first and last Mondays of the current month:

```
select next_day(trunc(sysdate,'mm')-1,'MONDAY') first_monday,
       next_day(last_day(trunc(sysdate,'mm'))-7,'MONDAY') last_monday
  from dual
```

PostgreSQL

Use the function DATE_TRUNC to find the first day of the month. Once you have the first day of the month, you can use simple arithmetic involving the numeric values of weekdays (Sun–Sat is 1–7) to find the first and last Mondays of the current month:

```
 1 select first_monday,
 2        case to_char(first_monday+28,'mm')
 3             when mth then first_monday+28
 4                      else first_monday+21
 5        end as last_monday
 6   from (
 7 select case sign(cast(to_char(dy,'d') as integer)-2)
 8             when 0
 9             then dy
10             when -1
11             then dy+abs(cast(to_char(dy,'d') as integer)-2)
12             when 1
13             then (7-(cast(to_char(dy,'d') as integer)-2))+dy
14        end as first_monday,
15        mth
16   from (
17 select cast(date_trunc('month',current_date) as date) as dy,
18        to_char(current_date,'mm') as mth
19   from t1
20        ) x
21        ) y
```

MySQL

Use the ADDDATE function to find the first day of the month. Once you have the first day of the month, you can use simple arithmetic on the numeric values of weekdays (Sun–Sat is 1–7) to find the first and last Mondays of the current month:

```
 1 select first_monday,
 2        case month(adddate(first_monday,28))
 3             when mth then adddate(first_monday,28)
 4                      else adddate(first_monday,21)
 5        end last_monday
 6  from (
 7 select case sign(dayofweek(dy)-2)
 8             when 0 then dy
 9             when -1 then adddate(dy,abs(dayofweek(dy)-2))
10             when 1 then adddate(dy,(7-(dayofweek(dy)-2)))
11        end first_monday,
12        mth
13   from (
```

```
select adddate(adddate(current_date,-day(current_date)),1) dy,
       month(current_date) mth
  from t1
       ) x
       ) y
```

SQL Server

Use the recursive WITH clause to generate each day in the current month, and then use a CASE expression to flag all Mondays. The first and last Mondays will be the earliest and latest of the flagged dates:

```
  with x (dy,mth,is_monday)
    as (
select dy,mth,
       case when datepart(dw,dy) = 2
            then 1 else 0
       end
  from (
select dateadd(day,1,dateadd(day,-day(getdate()),getdate())) dy,
       month(getdate()) mth
  from t1
       ) tmp1
 union all
select dateadd(day,1,dy),
       mth,
       case when datepart(dw,dateadd(day,1,dy)) = 2
            then 1 else 0
       end
  from x
 where month(dateadd(day,1,dy)) = mth
)
select min(dy) first_monday,
       max(dy) last_monday
  from x
 where is_monday = 1
```

Discussion

DB2 and SQL Server

DB2 and SQL Server use different functions to solve this problem, but the technique is exactly the same. If you eyeball both solutions, you'll see the only difference between the two is the way dates are added. This discussion will cover both solutions, using the DB2 solution's code to show the results of intermediate steps.

If you do not have access to the recursive WITH clause in the version of SQL Server or DB2 that you are running, you can use the PostgreSQL technique instead.

The first step in finding the first and last Mondays of the current month is to return the first day of the month. Inline view TMP1 in recursive view X finds the first day of the current month by first finding the current date, specifically, the day of the month for the current date. The day of the month for the current date represents how many days into the month you are (e.g., April 10th is the 10th day of the April). If you subtract this day of the month value from the current date, you end up at the last day of the previous month (e.g., subtracting 10 from April 10th puts you at the last day of March). After this subtraction, simply add one day to arrive at the first day of the current month:

```
select (current_date-day(current_date) day +1 day) dy
  from t1

DY
-----------
01-JUN-2005
```

Next, find the month for the current date using the MONTH function and a simple CASE expression to determine whether the first day of the month is a Monday:

```
select dy, month(dy) mth,
        case when dayname(dy)='Monday'
             then 1 else 0
        end is_monday
  from  (
select  (current_date-day(current_date) day +1 day) dy
  from  t1
        ) tmp1

DY          MTH  IS_MONDAY
----------- --- ----------
01-JUN-2005   6          0
```

Then use the recursive capabilities of the WITH clause to repeatedly add one day to the first day of the month until you're no longer in the current month. Along the way, you will use a CASE expression to determine which days in the month are Mondays (Mondays will be flagged with 1). A portion of the output from recursive view X is shown here:

```
with x (dy,mth,is_monday)
     as (
 select dy,month(dy) mth,
        case when dayname(dy)='Monday'
             then 1 else 0
```

```
        end is_monday
   from (
 select (current_date-day(current_date) day +1 day) dy
   from t1
        ) tmp1
  union all
 select (dy +1 day), mth,
        case when dayname(dy +1 day)='Monday'
             then 1 else 0
        end
   from x
  where month(dy +1 day) = mth
 )
 select *
   from x

DY          MTH  IS_MONDAY
----------- --- ----------
01-JUN-2005   6          0
02-JUN-2005   6          0
03-JUN-2005   6          0
04-JUN-2005   6          0
05-JUN-2005   6          0
06-JUN-2005   6          1
07-JUN-2005   6          0
08-JUN-2005   6          0
…
```

Only Mondays will have a value of 1 for IS_MONDAY, so the final step is to use the aggregate functions MIN and MAX on rows where IS_MONDAY is 1 to find the first and last Mondays of the month.

Oracle

The function NEXT_DAY makes this problem easy to solve. To find the first Monday of the current month, first return the last day of the prior month via some date arithmetic involving the TRUNC function:

```
select trunc(sysdate,'mm')-1 dy
  from dual

DY
-----------
31-MAY-2005
```

Then use the NEXT_DAY function to find the first Monday that comes after the last day of the previous month (i.e., the first Monday of the current month):

```
select next_day(trunc(sysdate,'mm')-1,'MONDAY') first_monday
  from dual
```

```
FIRST_MONDAY
------------
06-JUN-2005
```

To find the last Monday of the current month, start by returning the first day of the current month by using the TRUNC function:

```
select trunc(sysdate,'mm') dy
  from dual

DY
-----------
01-JUN-2005
```

The next step is to find the last week (the last seven days) of the month. Use the LAST_DAY function to find the last day of the month, and then subtract seven days:

```
select last_day(trunc(sysdate,'mm'))-7 dy
  from dual

DY
-----------
23-JUN-2005
```

If it isn't immediately obvious, you go back seven days from the last day of the month to ensure that you will have at least one of any weekday left in the month. The last step is to use the function NEXT_DAY to find the next (and last) Monday of the month:

```
select next_day(last_day(trunc(sysdate,'mm'))-7,'MONDAY') last_monday
  from dual

LAST_MONDAY
-----------
27-JUN-2005
```

PostgreSQL and MySQL

PostgreSQL and MySQL also share the same solution approach. The difference is in the functions that you invoke. Despite their lengths, the respective queries are extremely simple; little overhead is involved in finding the first and last Mondays of the current month.

The first step is to find the first day of the current month. The next step is to find the first Monday of the month. Since there is no function to find the next date for a given weekday, you need to use a little arithmetic. The CASE expression beginning on line 7 (of either solution) evaluates the difference between the numeric value for the weekday of the first day of the month and the numeric value corresponding to Monday. Given that the function TO_CHAR (PostgreSQL), when called with the *D* or *d* format, and the function DAYOFWEEK (MySQL) will return a numeric value from 1 to 7 representing days Sunday to Saturday, Monday is always represented by 2. The

first test evaluated by CASE is the SIGN of the numeric value of the first day of the month (whatever it may be) minus the numeric value of Monday (2). If the result is zero, then the first day of the month falls on a Monday, and that is the first Monday of the month. If the result is –1, then the first day of the month falls on a Sunday, and to find the first Monday of the month, simply add the difference in days between 2 and 1 (numeric values of Monday and Sunday, respectively) to the first day of the month.

If you are having trouble understanding how this works, forget the weekday names and just do the math. For example, say you happen to be starting on a Tuesday and you are looking for the next Friday. When using TO_CHAR with the *d* format, or DAYOFWEEK, Friday is 6 and Tuesday is 3. To get to 6 from 3, simply take the difference (6–3 = 3) and add it to the smaller value ((6–3) + 3 = 6). So, regardless of the actual dates, if the numeric value of the day you are starting from is less than the numeric value of the day you are searching for, adding the difference between the two dates to the date you are starting from will get you to the date you are searching for.

If the result from SIGN is 1, then the first day of the month falls between Tuesday and Saturday (inclusive). When the first day of the month has a numeric value greater than 2 (Monday), subtract from 7 the difference between the numeric value of the first day of the month and the numeric value of Monday (2), and then add that value to the first day of the month. You will have arrived at the day of the week that you are after, in this case Monday.

Again, if you are having trouble understanding how this works, forget the weekday names and just do the math. For example, suppose you want to find the next Tuesday and you are starting from Friday. Tuesday (3) is less than Friday (6). To get to 3 from 6, subtract the difference between the two values from 7 (7–(|3–6|) = 4) and add the result (4) to the start day Friday. (The vertical bars in |3–6| generate the absolute value of that difference.) Here, you're not adding 4 to 6 (which will give you 10); you are adding four days to Friday, which will give you the next Tuesday.

The idea behind the CASE expression is to create a sort of a "next day" function for PostgreSQL and MySQL. If you do not start with the first day of the month, the value for DY will be the value returned by CURRENT_DATE, and the result of the CASE expression will return the date of the next Monday starting from the current date (unless CURRENT_DATE is a Monday, then that date will be returned).

Now that you have the first Monday of the month, add either 21 or 28 days to find the last Monday of the month. The CASE expression in lines 2–5 determines whether to add 21 or 28 days by checking to see whether 28 days takes you into the next month. The CASE expression does this through the following process:

1. It adds 28 to the value of FIRST_MONDAY.
2. Using either TO_CHAR (PostgreSQL) or MONTH, the CASE expression extracts the name of the current month from the result of FIRST_MONDAY + 28.
3. The result from step two is compared to the value MTH from the inline view. The value MTH is the name of the current month as derived from CURRENT_DATE. If the 2 month values match, then the month is large enough for you to need to add 28 days, and the CASE expression returns FIRST_MONDAY + 28. If the two month values do not match, then you do not have room to add 28 days, and the CASE expression returns FIRST_MONDAY + 21 days instead. It is convenient that our months are such that 28 and 21 are the only two possible values you need worry about adding.

You can extend the solution by adding 7 and 14 days to find the second and third Mondays of the month, respectively.

9.7 Creating a Calendar

Problem

You want to create a calendar for the current month. The calendar should be formatted like a calendar you might have on your desk: seven columns across and (usually) five rows down.

Solution

Each solution will look a bit different, but they all solve the problem the same way: return each day for the current month, and then pivot on the day of the week for each week in the month to create a calendar.

There are different formats available for calendars. For example, the Unix CAL command formats the days from Sunday to Saturday. The examples in this recipe are based on ISO weeks, so the Monday through Friday format is the most convenient to generate. Once you become comfortable with the solutions, you'll see that

reformatting however you like is simply a matter of modifying the values assigned by the ISO week before pivoting.

As you begin to use different types of formatting with SQL to create readable output, you will notice your queries becoming longer. Don't let those long queries intimidate you; the queries presented for this recipe are extremely simple once broken down and run piece by piece.

DB2

Use the recursive WITH clause to return every day in the current month. Then pivot on the day of the week using CASE and MAX:

```
 with x(dy,dm,mth,dw,wk)
 as (
select (current_date -day(current_date) day +1 day) dy,
        day((current_date -day(current_date) day +1 day)) dm,
        month(current_date) mth,
        dayofweek(current_date -day(current_date) day +1 day) dw,
        week_iso(current_date -day(current_date) day +1 day) wk
 from t1
union all
select dy+1 day, day(dy+1 day), mth,
        dayofweek(dy+1 day), week_iso(dy+1 day)
 from x
where month(dy+1 day) = mth
)
select max(case dw when 2 then dm end) as Mo,
        max(case dw when 3 then dm end) as Tu,
        max(case dw when 4 then dm end) as We,
        max(case dw when 5 then dm end) as Th,
        max(case dw when 6 then dm end) as Fr,
        max(case dw when 7 then dm end) as Sa,
        max(case dw when 1 then dm end) as Su
 from x
group by wk
order by wk
```

Oracle

Use the recursive CONNECT BY clause to return each day in the current month. Then pivot on the day of the week using CASE and MAX:

```
with x
  as (
select *
 from (
select to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
```

```
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
       )
 where curr_mth = mth
)
select max(case dw when 2 then dm end) Mo,
       max(case dw when 3 then dm end) Tu,
       max(case dw when 4 then dm end) We,
       max(case dw when 5 then dm end) Th,
       max(case dw when 6 then dm end) Fr,
       max(case dw when 7 then dm end) Sa,
       max(case dw when 1 then dm end) Su
  from x
 group by wk
 order by wk
```

PostgreSQL

Use the function GENERATE_SERIES to return every day in the current month. Then pivot on the day of the week using MAX and CASE:

```
select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from (
select *
  from (
select cast(date_trunc('month',current_date) as date)+x.id,
       to_char(
          cast(
    date_trunc('month',current_date)
               as date)+x.id,'iw') as wk,
       to_char(
          cast(
    date_trunc('month',current_date)
               as date)+x.id,'dd') as dm,
       cast(
    to_char(
       cast(
  date_trunc('month',current_date)
                as date)+x.id,'d') as integer) as dw,
        to_char(
           cast(
    date_trunc('month',current_date)
                as date)+x.id,'mm') as curr_mth,
        to_char(current_date,'mm') as mth
```

```
30   from generate_series (0,31) x(id)
31        ) x
32  where mth = curr_mth
33        ) y
34  group by wk
35  order by wk
```

MySQL

Use a recursive CTE to return each day in the current month. Then pivot on the day of the week using MAX and CASE:

```
with recursive  x(dy,dm,mth,dw,wk)
      as (
  select dy,
         day(dy) dm,
         datepart(m,dy) mth,
         datepart(dw,dy) dw,
         case when datepart(dw,dy) = 1
              then datepart(ww,dy)-1
              else datepart(ww,dy)
        end wk
   from (
 select date_add(day,-day(getdate())+1,getdate()) dy
   from t1
        ) x
  union all
  select dateadd(d,1,dy), day(date_add(d,1,dy)), mth,
         datepart(dw,dateadd(d,1,dy)),
         case when datepart(dw,date_add(d,1,dy)) = 1
              then datepart(wk,date_add(d,1,dy))-1
              else datepart(wk,date_add(d,1,dy))
         end
    from x
   where datepart(m,date_add(d,1,dy)) = mth
 )
 select max(case dw when 2 then dm end) as Mo,
        max(case dw when 3 then dm end) as Tu,
        max(case dw when 4 then dm end) as We,
        max(case dw when 5 then dm end) as Th,
        max(case dw when 6 then dm end) as Fr,
        max(case dw when 7 then dm end) as Sa,
        max(case dw when 1 then dm end) as Su
   from x
  group by wk
  order by wk;
```

SQL Server

Use the recursive WITH clause to return every day in the current month. Then pivot on the day of the week using CASE and MAX:

```
 1   with x(dy,dm,mth,dw,wk)
 2     as (
 3 select dy,
 4        day(dy) dm,
 5        datepart(m,dy) mth,
 6        datepart(dw,dy) dw,
 7        case when datepart(dw,dy) = 1
 8             then datepart(ww,dy)-1
 9             else datepart(ww,dy)
10        end wk
11   from (
12 select dateadd(day,-day(getdate())+1,getdate()) dy
13   from t1
14        ) x
15  union all
16  select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
17         datepart(dw,dateadd(d,1,dy)),
18         case when datepart(dw,dateadd(d,1,dy)) = 1
19              then datepart(wk,dateadd(d,1,dy))  -1
20              else datepart(wk,dateadd(d,1,dy))
21         end
22    from x
23   where datepart(m,dateadd(d,1,dy)) = mth
24 )
25 select max(case dw when 2 then dm end) as Mo,
26        max(case dw when 3 then dm end) as Tu,
27        max(case dw when 4 then dm end) as We,
28        max(case dw when 5 then dm end) as Th,
29        max(case dw when 6 then dm end) as Fr,
30        max(case dw when 7 then dm end) as Sa,
31        max(case dw when 1 then dm end) as Su
32   from x
33  group by wk
34  order by wk
```

Discussion

DB2

The first step is to return each day in the month for which you want to create a calendar. Do that using the recursive WITH clause. Along with each day of the month (DM), you will need to return different parts of each date: the day of the week (DW), the current month you are working with (MTH), and the ISO week for each day of

the month (WK). The results of the recursive view X prior to recursion taking place (the upper portion of the UNION ALL) are shown here:

```
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2005 01  06          4 22
```

The next step is to repeatedly increase the value for DM (move through the days of the month) until you are no longer in the current month. As you move through each day in the month, you will also return the day of the week that each day is, and which ISO week the current day of the month falls into. Partial results are shown here:

```
with x(dy,dm,mth,dw,wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
 union all
 select dy+1 day, day(dy+1 day), mth,
        dayofweek(dy+1 day), week_iso(dy+1 day)
   from x
  where month(dy+1 day) = mth
)
select *
  from x

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2020 01 06           4 22
02-JUN-2020 02 06           5 22
…
21-JUN-2020 21 06           3 25
22-JUN-2020 22 06           4 25
…
30-JUN-2020 30 06           5 26
```

What you are returning at this point is: each day for the current month, the two-digit numeric day of the month, the two-digit numeric month, the one-digit day of the week (1–7 for Sun–Sat), and the two-digit ISO week each day falls into. With all this information available, you can use a CASE expression to determine which day of the

week each value of DM (each day of the month) falls into. A portion of the results is shown here:

```
with x(dy,dm,mth,dw,wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
 union all
 select dy+1 day, day(dy+1 day), mth,
        dayofweek(dy+1 day), week_iso(dy+1 day)
   from x
  where month(dy+1 day) = mth
 )
 select wk,
        case dw when 2 then dm end as Mo,
        case dw when 3 then dm end as Tu,
        case dw when 4 then dm end as We,
        case dw when 5 then dm end as Th,
        case dw when 6 then dm end as Fr,
        case dw when 7 then dm end as Sa,
        case dw when 1 then dm end as Su
   from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
23       08
23          09
23             10
23                11
23                   12
```

As you can see from the partial output, every day in each week is returned as a row. What you want to do now is to group the days by week, and then collapse all the days for each week into a single row. Use the aggregate function MAX, and group by WK (the ISO week) to return all the days for a week as one row. To properly format the calendar and ensure that the days are in the right order, order the results by WK. The final output is shown here:

```
with x(dy,dm,mth,dw,wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
```

```
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
 union all
 select dy+1 day, day(dy+1 day), mth,
        dayofweek(dy+1 day), week_iso(dy+1 day)
   from x
  where month(dy+1 day) = mth
)
select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from x
 group by wk
 order by wk
```

```
MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Oracle

Begin by using the recursive CONNECT BY clause to generate a row for each day in the month for which you want to generate a calendar. If you aren't running at least Oracle9*i* Database, you can't use CONNECT BY this way. Instead, you can use a pivot table, such as T500 in the MySQL solution.

Along with each day of the month, you will need to return different bits of information for each day: the day of the month (DM), the day of the week (DW), the current month you are working with (MTH), and the ISO week for each day of the month (WK). The results of the WITH view X for the first day of the current month are shown here:

```
select trunc(sysdate,'mm') dy,
       to_char(trunc(sysdate,'mm'),'dd') dm,
       to_char(sysdate,'mm') mth,
       to_number(to_char(trunc(sysdate,'mm'),'d')) dw,
       to_char(trunc(sysdate,'mm'),'iw') wk
  from dual
```

```
DY          DM MT         DW WK
----------- -- -- ---------- --
01-JUN-2020 01 06          4 22
```

The next step is to repeatedly increase the value for DM (move through the days of the month) until you are no longer in the current month. As you move through each day in the month, you will also return the day of the week for each day and the ISO week into which the current day falls. Partial results are shown here (the full date for each day is added for readability):

```
with x
  as (
select *
  from (
select trunc(sysdate,'mm')+level-1 dy,
       to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
       )
 where curr_mth = mth
)
select *
  from x

DY          WK DM         DW CU MT
----------- -- -- ---------- -- --
01-JUN-2020 22 01          4 06 06
02-JUN-2020 22 02          5 06 06
…
21-JUN-2020 25 21          3 06 06
22-JUN-2020 25 22          4 06 06
…
30-JUN-2020 26 30          5 06 06
```

What you are returning at this point is one row for each day of the current month. In that row you have: the two-digit numeric day of the month, the two-digit numeric month, the one-digit day of the week (1–7 for Sun–Sat), and the two-digit ISO week number. With all this information available, you can use a CASE expression to determine which day of the week each value of DM (each day of the month) falls into. A portion of the results is shown here:

```
with x
  as (
select *
  from (
select trunc(sysdate,'mm')+level-1 dy,
       to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
```

```
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
       )
 where curr_mth = mth
)
select wk,
       case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                  05
23 06
23    07
23       08
23          09
23             10
23               11
23                 12
```

As you can see from the partial output, every day in each week is returned as a row, but the day number is in one of seven columns corresponding to the day of the week. Your task now is to consolidate the days into one row for each week. Use the aggregate function MAX and group by WK (the ISO week) to return all the days for a week as one row. To ensure the days are in the right order, order the results by WK. The final output is shown here:

```
with x
  as (
select *
  from (
select to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
```

```
       )
 where curr_mth = mth
)
select max(case dw when 2 then dm end) Mo,
       max(case dw when 3 then dm end) Tu,
       max(case dw when 4 then dm end) We,
       max(case dw when 5 then dm end) Th,
       max(case dw when 6 then dm end) Fr,
       max(case dw when 7 then dm end) Sa,
       max(case dw when 1 then dm end) Su
  from x
 group by wk
 order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

MySQL, PostgreSQL, and SQL Server

These solutions are the same except for differences in the specific functions used to call dates. We arbitrarily use the SQL Serve solution for the explanation. Begin by returning one row for each day of the month. You can do that using the recursive WITH clause. For each row that you return, you will need the following items: the day of the month (DM), the day of the week (DW), the current month you are working with (MTH), and the ISO week for each day of the month (WK). The results of the recursive view X prior to recursion taking place (the upper portion of the UNION ALL) are shown here:

```
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2005  1   6          4 23
```

Your next step is to repeatedly increase the value for DM (move through the days of the month) until you are no longer in the current month. As you move through each day in the month, you will also return the day of the week and the ISO week number. Partial results are shown here:

```
  with x(dy,dm,mth,dw,wk)
    as (
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x
 union all
 select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
        datepart(dw,dateadd(d,1,dy)),
        case when datepart(dw,dateadd(d,1,dy)) = 1
             then datepart(wk,dateadd(d,1,dy))-1
             else datepart(wk,dateadd(d,1,dy))
        end
  from x
 where datepart(m,dateadd(d,1,dy)) = mth
)
select *
  from x

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2005 01 06           4 23
02-JUN-2005 02 06           5 23
…
21-JUN-2005 21 06           3 26
22-JUN-2005 22 06           4 26
…
30-JUN-2005 30 06           5 27
```

For each day in the current month, you now have: the two-digit numeric day of the month, the two-digit numeric month, the one-digit day of the week (1–7 for Sun–Sat), and the two-digit ISO week number.

Now, use a CASE expression to determine which day of the week each value of DM (each day of the month) falls into. A portion of the results is shown here:

```
  with x(dy,dm,mth,dw,wk)
    as (
select dy,
```

```
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x
 union all
 select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
        datepart(dw,dateadd(d,1,dy)),
        case when datepart(dw,dateadd(d,1,dy)) = 1
             then datepart(wk,dateadd(d,1,dy))-1
             else datepart(wk,dateadd(d,1,dy))
        end
   from x
  where datepart(m,dateadd(d,1,dy)) = mth
)
select case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
23       08
23          09
23             10
23                11
23                   12
```

Every day in each week is returned as a separate row. In each row, the column containing the day number corresponds to the day of the week. You now need to consolidate the days for each week into one row. Do that by grouping the rows by WK (the ISO week) and applying the MAX function to the different columns. The results will be in calendar format as shown here:

```
with x(dy,dm,mth,dw,wk)
    as (
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x
 union all
 select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
        datepart(dw,dateadd(d,1,dy)),
        case when datepart(dw,dateadd(d,1,dy)) = 1
             then datepart(wk,dateadd(d,1,dy))-1
             else datepart(wk,dateadd(d,1,dy))
        end
   from x
  where datepart(m,dateadd(d,1,dy)) = mth
)
select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from x
 group by wk
 order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

9.8 Listing Quarter Start and End Dates for the Year

Problem

You want to return the start and end dates for each of the four quarters of a given year.

Solution

There are four quarters to a year, so you know you will need to generate four rows. After generating the desired number of rows, simply use the date functions supplied by your RDBMS to return to the quarter the start and end dates fall into. Your goal is to produce the following result set (one again, the choice to use the current year is arbitrary):

```
QTR Q_START     Q_END
--- ----------- -----------
  1 01-JAN-2020 31-MAR-2020
  2 01-APR-2020 30-JUN-2020
  3 01-JUL-2020 30-SEP-2020
  4 01-OCT-2020 31-DEC-2020
```

DB2

Use table EMP and the window function ROW_NUMBER OVER to generate four rows. Alternatively, you can use the WITH clause to generate rows (as many of the recipes do), or you can query against any table with at least four rows. The following solution uses the ROW_NUMBER OVER approach:

```
 1 select quarter(dy-1 day) QTR,
 2        dy-3 month Q_start,
 3        dy-1 day Q_end
 4   from (
 5 select (current_date -
 6          (dayofyear(current_date)-1) day
 7            + (rn*3) month) dy
 8   from (
 9 select row_number()over() rn
10   from emp
11  fetch first 4 rows only
12         ) x
13         ) y
```

Oracle

Use the function ADD_MONTHS to find the start and end dates for each quarter. Use ROWNUM to represent the quarter the start and end dates belong to. The following solution uses table EMP to generate four rows:

```
1 select rownum qtr,
2        add_months(trunc(sysdate,'y'),(rownum-1)*3) q_start,
3        add_months(trunc(sysdate,'y'),rownum*3)-1 q_end
4   from emp
5  where rownum <= 4
```

PostgreSQL

Find the first day of the year based on the current date, and use a recursive CTE to fill in the first date of the remaining three quarters before finding the last day of each quarter:

```
 with recursive x (dy,cnt)
    as (
 select
       current_date -cast(extract(day from current_date)as integer) +1 dy
       , id
   from t1
  union all
 select cast(dy  + interval '3 months' as date) , cnt+1
   from x
  where cnt+1 <= 4
)
select  cast(dy - interval '3 months' as date) as Q_start
       , dy-1 as Q_end
               from x
```

MySQL

Find the first day of the year from the current day, and use a CTE to create four rows, one for each quarter. Use ADDDATE to find the last day of each quarter (three months after the previous last day, or the first day of the quarter minus one):

```
            with recursive x (dy,cnt)
    as (
               select
        adddate(current_date,(-dayofyear(current_date))+1) dy
          ,id
           from t1
         union all
               select adddate(dy, interval 3 month ), cnt+1
               from x
        where cnt+1 <= 4
       )

      select quarter(adddate(dy,-1)) QTR
   ,  date_add(dy, interval -3 month) Q_start
   ,  adddate(dy,-1)  Q_end
    from x
    order by 1;
```

SQL Server

Use the recursive WITH clause to generate four rows. Use the function DATEADD to find the start and end dates. Use the function DATEPART to determine which quarter the start and end dates belong to:

```
 1  with x (dy,cnt)
 2    as (
 3 select dateadd(d,-(datepart(dy,getdate())-1),getdate()),
 4        1
 5   from t1
 6  union all
 7 select dateadd(m,3,dy), cnt+1
 8   from x
 9  where cnt+1 <= 4
10 )
11 select datepart(q,dateadd(d,-1,dy)) QTR,
12         dateadd(m,-3,dy) Q_start,
13        dateadd(d,-1,dy) Q_end
14   from x
15 order by 1
```

Discussion

DB2

The first step is to generate four rows (with values one through four) for each quarter in the year. Inline view X uses the window function ROW_NUMBER OVER and the FETCH FIRST clause to return only four rows from EMP. The results are shown here:

```
select row_number()over() rn
  from emp
 fetch first 4 rows only

RN
--
 1
 2
 3
 4
```

The next step is to find the first day of the year, then add *n* months to it, where *n* is three times RN (you are adding 3, 6, 9, and 12 months to the first day of the year). The results are shown here:

```
select (current_date
        (dayofyear(current_date)-1) day
          + (rn*3) month) dy
  from (
select row_number()over() rn
  from emp
 fetch first 4 rows only
       ) x

DY
-----------
01-APR-2005
01-JUL-2005
```

```
01-OCT-2005
01-JAN-2005
```

At this point, the values for DY are one day after the end date for each quarter. The next step is to get the start and end dates for each quarter. Subtract one day from DY to get the end of each quarter, and subtract three months from DY to get the start of each quarter. Use the QUARTER function on DY-1 (the end date for each quarter) to determine which quarter the start and end dates belong to.

Oracle

The combination of ROWNUM, TRUNC, and ADD_MONTHS makes this solution easy. To find the start of each quarter, simply add *n* months to the first day of the year, where *n* is (ROWNUM-1)*3 (giving you 0, 3, 6, 9). To find the end of each quarter, add *n* months to the first day of the year, where *n* is ROWNUM*3, and subtract one day. As an aside, when working with quarters, you may also find it useful to use TO_CHAR and/or TRUNC with the Q formatting option.

PostgreSQL, MySQL, and SQL Server

Like some of the previous recipes, this recipe uses the same structure across three RDMS implementations, but different syntax for the date operations. The first step is to find the first day of the year and then recursively add *n* months, where *n* is three times the current iteration (there are four iterations, therefore, you are adding 3*1 months, 3*2 months, etc.), using the DATEADD function or its equivalent. The results are shown here:

```
with x (dy,cnt)
   as (
select dateadd(d,-(datepart(dy,getdate())-1),getdate()),
       1
  from t1
 union all
select dateadd(m,3,dy), cnt+1
  from x
 where cnt+1 <= 4
)
select dy
  from x

DY
-----------
01-APR-2020
01-JUL-2020
01-OCT-2020
01-JAN-2020
```

The values for DY are one day after the end of each quarter. To get the end of each quarter, simply subtract one day from DY by using the DATEADD function. To find

the start of each quarter, use the DATEADD function to subtract three months from DY. Use the DATEPART function on the end date for each quarter to determine which quarter the start and end dates belong to or its equivalent. If you are using PostgreSQL, note that you need CAST to ensure data types align after performing adding the three months to the start date, or the data types will different, and the UNION ALL in the recursive CTE will fail.

9.9 Determining Quarter Start and End Dates for a Given Quarter

Problem

When given a year and quarter in the format of YYYYQ (four-digit year, one-digit quarter), you want to return the quarter's start and end dates.

Solution

The key to this solution is to find the quarter by using the modulus function on the YYYYQ value. (As an alternative to modulo, since the year format is four digits, you can simply substring out the last digit to get the quarter.) Once you have the quarter, simply multiply by three to get the ending month for the quarter. In the solutions that follow, inline view X will return all four year and quarter combinations. The result set for inline view X is as follows:

```
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
select 20054 as yrq from t1
    YRQ
-------
  20051
  20052
  20053
  20054
```

DB2

Use the function SUBSTR to return the year from inline view X. Use the MOD function to determine which quarter you are looking for:

```
 1 select (q_end-2 month) q_start,
 2        (q_end+1 month)-1 day q_end
 3   from (
 4 select date(substr(cast(yrq as char(4)),1,4) ||'-'||
 5        rtrim(cast(mod(yrq,10)*3 as char(2))) ||'-1') q_end
 6   from (
 7 select 20051 yrq from t1 union all
```

```
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x
       ) y
```

Oracle

Use the function SUBSTR to return the year from inline view X. Use the MOD function to determine which quarter you are looking for:

```
select add_months(q_end,-2) q_start,
       last_day(q_end) q_end
  from (
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x
       ) y
```

PostgreSQL

Use the function SUBSTR to return the year from the inline view X. Use the MOD function to determine which quarter you are looking for:

```
select date(q_end-(2*interval '1 month')) as q_start,
       date(q_end+interval '1 month'-interval '1 day') as q_end
  from (
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') as q_end
  from (
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
select 20054 as yrq from t1
       ) x
       ) y
```

MySQL

Use the function SUBSTR to return the year from the inline view X. Use the MOD function to determine which quarter you are looking for:

```
select date_add(
        adddate(q_end,-day(q_end)+1),
               interval -2 month) q_start,
       q_end
  from (
select last_day(
    str_to_date(
```

```
 8          concat(
 9          substr(yrq,1,4),mod(yrq,10)*3),'%Y%m')) q_end
10   from (
11 select 20051 as yrq from t1 union all
12 select 20052 as yrq from t1 union all
13 select 20053 as yrq from t1 union all
14 select 20054 as yrq from t1
15        ) x
16        ) y
```

SQL Server

Use the function SUBSTRING to return the year from the inline view X. Use the modulus function (%) to determine which quarter you are looking for:

```
 1 select dateadd(m,-2,q_end) q_start,
 2        dateadd(d,-1,dateadd(m,1,q_end)) q_end
 3   from (
 4 select cast(substring(cast(yrq as varchar),1,4)+'-'+
 5        cast(yrq%10*3 as varchar)+'-1' as datetime) q_end
 6   from (
 7 select 20051 as yrq from t1 union all
 8 select 20052 as yrq from t1 union all
 9 select 20052 as yrq from t1 union all
10 select 20054 as yrq from t1
11        ) x
12        ) y
```

Discussion

DB2

The first step is to find the year and quarter you are working with. Substring out the year from inline view X (X.YRQ) using the SUBSTR function. To get the quarter, use modulus 10 on YRQ. Once you have the quarter, multiply by three to get the end month for the quarter. The results are shown here:

```
select substr(cast(yrq as char(4)),1,4) yr,
       mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

YR      MTH
---- ------
2005      3
2005      6
```

```
2005      9
2005     12
```

At this point you have the year and end month for each quarter. Use those values to construct a date, specifically, the first day of the last month for each quarter. Use the concatenation operator || to glue together the year and month, and then use the DATE function to convert to a date:

```
select date(substr(cast(yrq as char(4)),1,4) ||'-'||
       rtrim(cast(mod(yrq,10)*3 as char(2))) ||'-1') q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

The values for Q_END are the first day of the last month of each quarter. To get to the last day of the month, add one month to Q_END and then subtract one day. To find the start date for each quarter, subtract two months from Q_END.

Oracle

The first step is to find the year and quarter you are working with. Substring out the year from inline view X (X.YRQ) using the SUBSTR function. To get the quarter, use modulus 10 on YRQ. Once you have the quarter, multiply by three to get the end month for the quarter. The results are shown here:

```
select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

YR      MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

At this point you have the year and end month for each quarter. Use those values to construct a date, specifically, the first day of the last month for each quarter. Use the concatenation operator || to glue together the year and month, and then use the TO_DATE function to convert to a date:

```
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

The values for Q_END are the first day of the last month of each quarter. To get to the last day of the month, use the LAST_DAY function on Q_END. To find the start date for each quarter, subtract two months from Q_END using the ADD_MONTHS function.

PostgreSQL

The first step is to find the year and quarter you are working with. Substring out the year from inline view X (X.YRQ) using the SUBSTR function. To get the quarter, use modulus 10 on YRQ. Once you have the quarter, multiply by 3 to get the end month for the quarter. The results are shown here:

```
select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

YR        MTH
----  -------
2005        3
2005        6
2005        9
2005       12
```

At this point, you have the year and end month for each quarter. Use those values to construct a date, specifically, the first day of the last month for each quarter. Use the concatenation operator || to glue together the year and month, and then use the TO_DATE function to convert to a date:

```
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

The values for Q_END are the first day of the last month of each quarter. To get to the last day of the month, add one month to Q_END and subtract one day. To find the start date for each quarter, subtract two months from Q_END. Cast the final result as dates.

MySQL

The first step is to find the year and quarter you are working with. Substring out the year from inline view X (X.YRQ) using the SUBSTR function. To get the quarter, use modulus 10 on YRQ. Once you have the quarter, multiply by three to get the end month for the quarter. The results are shown here:

```
select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

YR      MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

At this point, you have the year and end month for each quarter. Use those values to construct a date, specifically, the last day of each quarter. Use the CONCAT function

to glue together the year and month, and then use the STR_TO_DATE function to convert to a date. Use the LAST_DAY function to find the last day for each quarter:

```
select last_day(
    str_to_date(
         concat(
         substr(yrq,1,4),mod(yrq,10)*3),'%Y%m')) q_end
  from (
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
select 20054 as yrq from t1
       ) x

Q_END
-----------
31-MAR-2005
30-JUN-2005
30-SEP-2005
31-DEC-2005
```

Because you already have the end of each quarter, all that's left is to find the start date for each quarter. Use the DAY function to return the day of the month the end of each quarter falls on, and subtract that from Q_END using the ADDDATE function to give you the end of the prior month; add one day to bring you to the first day of the last month of each quarter. The last step is to use the DATE_ADD function to subtract two months from the first day of the last month of each quarter to get you to the start date for each quarter.

SQL Server

The first step is to find the year and quarter you are working with. Substring out the year from inline view X (X.YRQ) using the SUBSTRING function. To get the quarter, use modulus 10 on YRQ. Once you have the quarter, multiply by three to get the end month for the quarter. The results are shown here:

```
select substring(yrq,1,4) yr, yrq%10*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

YR        MTH
----   ------
2005        3
2005        6
2005        9
2005       12
```

At this point, you have the year and end month for each quarter. Use those values to construct a date, specifically, the first day of the last month for each quarter. Use the concatenation operator + to glue together the year and month, and then use the CAST function to convert to a date:

```
select cast(substring(cast(yrq as varchar),1,4)+'-'+
       cast(yrq%10*3 as varchar)+'-1' as datetime) q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

The values for Q_END are the first day of the last month of each quarter. To get to the last day of the month, add one month to Q_END and subtract one day using the DATEADD function. To find the start date for each quarter, subtract two months from Q_END using the DATEADD function.

9.10 Filling in Missing Dates

Problem

You need to generate a row for every date (or every month, week, or year) within a given range. Such rowsets are often used to generate summary reports. For example, you want to count the number of employees hired every month of every year in which any employee has been hired. Examining the dates of all the employees hired, there have been hirings from 2000 to 2003:

```
select distinct
       extract(year from hiredate) as year
  from emp

YEAR
-----
 2000
 2001
 2002
 2003
```

You want to determine the number of employees hired each month from 2000 to 2003. A portion of the desired result set is shown here:

```
MTH          NUM_HIRED
----------- ----------
01-JAN-2001          0
01-FEB-2001          2
01-MAR-2001          0
01-APR-2001          1
01-MAY-2001          1
01-JUN-2001          1
01-JUL-2001          0
01-AUG-2001          0
01-SEP-2001          2
01-OCT-2001          0
01-NOV-2001          1
01-DEC-2001          2
```

Solution

The trick here is that you want to return a row for each month even if no employee was hired (i.e., the count would be zero). Because there isn't an employee hired every month between 2000 and 2003, you must generate those months yourself and then outer join to table EMP on HIREDATE (truncating the actual HIREDATE to its month so it can match the generated months when possible).

DB2

Use the recursive WITH clause to generate every month (the first day of each month from January 1, 2000, to December 1, 2003). Once you have all the months for the required range of dates, outer join to table EMP and use the aggregate function COUNT to count the number of hires for each month:

```
 1   with x (start_date,end_date)
 2     as (
 3 select (min(hiredate)
 4         dayofyear(min(hiredate)) day +1 day) start_date,
 5        (max(hiredate)
 6         dayofyear(max(hiredate)) day +1 day) +1 year end_date
 7   from emp
 8  union all
 9 select start_date +1 month, end_date
10   from x
11  where (start_date +1 month) < end_date
12 )
13 select x.start_date mth, count(e.hiredate) num_hired
14   from x left join emp e
15     on (x.start_date = (e.hiredate-(day(hiredate)-1) day))
16  group by x.start_date
17  order by 1
```

Oracle

Use the CONNECT BY clause to generate each month between 2000 and 2003. Then outer join to table EMP and use the aggregate function COUNT to count the number of employees hired in each month:

```
  with x
    as (
select add_months(start_date,level-1) start_date
  from (
select min(trunc(hiredate,'y')) start_date,
       add_months(max(trunc(hiredate,'y')),12) end_date
  from emp
       )
 connect by level <= months_between(end_date,start_date)
)
select x.start_date MTH, count(e.hiredate) num_hired
  from x left join emp e
    on (x.start_date = trunc(e.hiredate,'mm'))
 group by x.start_date
 order by 1
```

PostgreSQL

Use CTE to fill in the months since the earliest hire and then LEFT OUTER JOIN on the EMP table using the month and year of each generated month to enable the COUNT of the number of hiredates in each period:

```
        with recursive x (start_date, end_date)
as
(
    select
    cast(min(hiredate) - (cast(extract(day from min(hiredate))
    as integer) - 1) as date)
    , max(hiredate)
    from emp
  union all
    select cast(start_date + interval '1 month' as date)
    , end_date
    from x
    where start_date < end_date
 )

 select x.start_date,count(hiredate)
 from x left join emp
  on (extract(month from start_date) =
                   extract(month from emp.hiredate)
          and extract(year from start_date)
          = extract(year from emp.hiredate))
         group by x.start_date
         order by 1
```

MySQL

Use a recursive CTE to generate each month between the start and end dates, and then check for hires by using an outer join to table EMP:

```
with recursive x (start_date,end_date)
            as
           (
     select
         adddate(min(hiredate),
         -dayofyear(min(hiredate))+1)  start_date
         ,adddate(max(hiredate),
         -dayofyear(max(hiredate))+1)  end_date
         from emp
      union all
         select date_add(start_date,interval 1 month)
         , end_date
         from x
         where date_add(start_date, interval 1 month) < end_date
     )

      select x.start_date mth, count(e.hiredate) num_hired
             from x left join emp e
             on (extract(year_month from start_date)
                 =
                 extract(year_month from e.hiredate))
             group by x.start_date
             order by 1;
```

SQL Server

Use the recursive WITH clause to generate every month (the first day of each month from January 1, 2000, to December 1, 2003). Once you have all the months for the required range of dates, outer join to table EMP and use the aggregate function COUNT to count the number of hires for each month:

```
1   with x (start_date,end_date)
2     as (
3 select (min(hiredate) -
4          datepart(dy,min(hiredate))+1) start_date,
5        dateadd(yy,1,
6         (max(hiredate) -
7          datepart(dy,max(hiredate))+1)) end_date
8   from emp
9  union all
10 select dateadd(mm,1,start_date), end_date
11   from x
12  where dateadd(mm,1,start_date) < end_date
13 )
14 select x.start_date mth, count(e.hiredate) num_hired
15   from x left join emp e
16     on (x.start_date =
```

```
17            dateadd(dd,-day(e.hiredate)+1,e.hiredate))
18 group by x.start_date
19 order by 1
```

Discussion

DB2

The first step is to generate every month (actually the first day of each month) from 2000 to 2003. Start using the DAYOFYEAR function on the MIN and MAX HIREDATEs to find the boundary months:

```
select (min(hiredate)
        dayofyear(min(hiredate)) day +1 day) start_date,
       (max(hiredate)
        dayofyear(max(hiredate)) day +1 day) +1 year end_date
  from emp

START_DATE  END_DATE
----------- -----------
01-JAN-2000 01-JAN-2004
```

Your next step is to repeatedly add months to START_DATE to return all the months necessary for the final result set. The value for END_DATE is one day more than it should be. This is OK. As you recursively add months to START_DATE, you can stop before you hit END_DATE. A portion of the months created is shown here:

```
with x (start_date,end_date)
  as (
select (min(hiredate)
        dayofyear(min(hiredate)) day +1 day) start_date,
       (max(hiredate)
        dayofyear(max(hiredate)) day +1 day) +1 year end_date
  from emp
 union all
select start_date +1 month, end_date
  from x
 where (start_date +1 month) < end_date
)
select *
  from x

START_DATE  END_DATE
----------- -----------
01-JAN-2000 01-JAN-2004
01-FEB-2000 01-JAN-2004
01-MAR-2000 01-JAN-2004
…
01-OCT-2003 01-JAN-2004
01-NOV-2003 01-JAN-2004
01-DEC-2003 01-JAN-2004
```

At this point, you have all the months you need, and you can simply outer join to EMP.HIREDATE. Because the day for each START_DATE is the first of the month, truncate EMP.HIREDATE to the first day of its month. Finally, use the aggregate function COUNT on EMP.HIREDATE.

Oracle

The first step is to generate the first day of every for every month from 2000 to 2003. Start by using TRUNC and ADD_MONTHS together with the MIN and MAX HIREDATE values to find the boundary months:

```
select min(trunc(hiredate,'y')) start_date,
       add_months(max(trunc(hiredate,'y')),12) end_date
  from emp

START_DATE  END_DATE
----------- -----------
01-JAN-2000 01-JAN-2004
```

Then repeatedly add months to START_DATE to return all the months necessary for the final result set. The value for END_DATE is one day more than it should be, which is OK. As you recursively add months to START_DATE, you can stop before you hit END_DATE. A portion of the months created is shown here:

```
with x as (
select add_months(start_date,level-1) start_date
  from (
select min(trunc(hiredate,'y')) start_date,
       add_months(max(trunc(hiredate,'y')),12) end_date
  from emp
       )
 connect by level <= months_between(end_date,start_date)
)
select *
  from x

START_DATE
-----------
01-JAN-2000
01-FEB-2000
01-MAR-2000
…
01-OCT-2003
01-NOV-2003
01-DEC-2003
```

At this point, you have all the months you need, and you can simply outer join to EMP.HIREDATE. Because the day for each START_DATE is the first of the month, truncate EMP.HIREDATE to the first day of the month it is in. The final step is to use the aggregate function COUNT on EMP.HIREDATE.

PostgreSQL

This solution uses a CTE to generate the months you need and is similar to the subsequent solutions for MySQL and SQL Server. The first step is to create the boundary dates using aggregate functions. You could simply find earliest and latest hire dates using the MIN() and MAX() functions, but the output makes more sense if you find the first day of the month containing the earliest hire date.

MySQL

First, find the boundary dates by using the aggregate functions MIN and MAX along with the DAYOFYEAR and ADDDATE functions. The result set shown here is from inline view X:

```
with recursive x (start_date,end_date)
            as (
          select
           adddate(min(hiredate),
          -dayofyear(min(hiredate))+1)  start_date
          ,adddate(max(hiredate),
          -dayofyear(max(hiredate))+1)  end_date
          from emp
       union all
       select date_add(start_date,interval 1 month)
       , end_date
       from x
       where date_add(start_date, interval 1 month) < end_date
         )
 select * from x
```

```
select adddate(min(hiredate),-dayofyear(min(hiredate))+1) min_hd,
       adddate(max(hiredate),-dayofyear(max(hiredate))+1) max_hd
  from emp

MIN_HD      MAX_HD
----------- -----------
01-JAN-2000 01-JAN-2003
```

Next, increment MAX_HD to the last month of the year by the CTE:

```
MTH
-----------
01-JAN-2000
01-FEB-2000
01-MAR-2000
…
01-OCT-2003
01-NOV-2003
01-DEC-2003
```

Now that you have all the months you need for the final result set, outer join to EMP.HIREDATE (be sure to truncate EMP.HIREDATE to the first day of the month)

and use the aggregate function COUNT on EMP.HIREDATE to count the number of hires in each month.

SQL Server

Begin by generating every month (actually, the first day of each month) from 2000 to 2003. Then find the boundary months by applying the DAYOFYEAR function to the MIN and MAX HIREDATEs:

```
select (min(hiredate) -
         datepart(dy,min(hiredate))+1) start_date,
       dateadd(yy,1,
        (max(hiredate) -
         datepart(dy,max(hiredate))+1)) end_date
  from emp

START_DATE  END_DATE
----------- -----------
01-JAN-2000 01-JAN-2004
```

Your next step is to repeatedly add months to START_DATE to return all the months necessary for the final result set. The value for END_DATE is one day more than it should be, which is OK, as you can stop recursively adding months to START_DATE before you hit END_DATE. A portion of the months created is shown here:

```
with x (start_date,end_date)
  as (
select (min(hiredate) -
         datepart(dy,min(hiredate))+1) start_date,
       dateadd(yy,1,
        (max(hiredate) -
         datepart(dy,max(hiredate))+1)) end_date
  from emp
 union all
select dateadd(mm,1,start_date), end_date
  from x
 where dateadd(mm,1,start_date) < end_date
)
select *
  from x

START_DATE  END_DATE
----------- -----------
01-JAN-2000 01-JAN-2004
01-FEB-2000 01-JAN-2004
01-MAR-2000 01-JAN-2004
…
01-OCT-2003 01-JAN-2004
01-NOV-2003 01-JAN-2004
01-DEC-2003 01-JAN-2004
```

At this point, you have all the months you need. Simply outer join to EMP.HIREDATE. Because the day for each START_DATE is the first of the month, truncate EMP.HIREDATE to the first day of the month. The final step is to use the aggregate function COUNT on EMP.HIREDATE.

9.11 Searching on Specific Units of Time

Problem

You want to search for dates that match a given month, day of the week, or some other unit of time. For example, you want to find all employees hired in February or December, as well as employees hired on a Tuesday.

Solution

Use the functions supplied by your RDBMS to find month and weekday names for dates. This particular recipe can be useful in various places. Consider, if you wanted to search HIREDATEs but wanted to ignore the year by extracting the month (or any other part of the HIREDATE you are interested in), you can do so. The example solutions to this problem search by month and weekday name. By studying the date formatting functions provided by your RDBMS, you can easily modify these solutions to search by year, quarter, combination of year and quarter, month and year combination, etc.

DB2 and MySQL

Use the functions MONTHNAME and DAYNAME to find the name of the month and weekday an employee was hired, respectively:

```
1 select ename
2   from emp
3 where monthname(hiredate) in ('February','December')
4    or dayname(hiredate) = 'Tuesday'
```

Oracle and PostgreSQL

Use the function TO_CHAR to find the names of the month and weekday an employee was hired. Use the function RTRIM to remove trailing whitespaces:

```
1 select ename
2   from emp
3 where rtrim(to_char(hiredate,'month')) in ('february','december')
4    or rtrim(to_char(hiredate,'day')) = 'tuesday'
```

SQL Server

Use the function DATENAME to find the names of the month and weekday an employee was hired:

```
1 select ename
2   from emp
3 where datename(m,hiredate) in ('February','December')
4    or datename(dw,hiredate) = 'Tuesday'
```

Discussion

The key to each solution is simply knowing which functions to use and how to use them. To verify what the return values are, put the functions in the SELECT clause and examine the output. Listed here is the result set for employees in DEPTNO 10 (using SQL Server syntax):

```
select ename,datename(m,hiredate) mth,datename(dw,hiredate) dw
  from emp
 where deptno = 10

ENAME   MTH        DW
------  ---------  -----------
CLARK   June       Tuesday
KING    November   Tuesday
MILLER  January    Saturday
```

Once you know what the function(s) return, finding rows using the functions shown in each of the solutions is easy.

9.12 Comparing Records Using Specific Parts of a Date

Problem

You want to find which employees have been hired on the same month and weekday. For example, if an employee was hired on Monday, March 10, 2008, and another employee was hired on Monday, March 2, 2001, you want those two to come up as a match since the day of week and month match. In table EMP, only three employees meet this requirement. You want to return the following result set:

```
MSG
-----------------------------------------------------
JAMES was hired on the same month and weekday as FORD
SCOTT was hired on the same month and weekday as JAMES
SCOTT was hired on the same month and weekday as FORD
```

Solution

Because you want to compare one employee's HIREDATE with the HIREDATE of the other employees, you will need to self-join table EMP. That makes each possible combination of HIREDATEs available for you to compare. Then, simply extract the weekday and month from each HIREDATE and compare.

DB2

After self-joining table EMP, use the function DAYOFWEEK to return the numeric day of the week. Use the function MONTHNAME to return the name of the month:

```
select a.ename ||
       ' was hired on the same month and weekday as '||
       b.ename msg
  from emp a, emp b
where (dayofweek(a.hiredate),monthname(a.hiredate)) =
      (dayofweek(b.hiredate),monthname(b.hiredate))
  and a.empno < b.empno
order by a.ename
```

Oracle and PostgreSQL

After self-joining table EMP, use the TO_CHAR function to format the HIREDATE into weekday and month for comparison:

```
select a.ename ||
       ' was hired on the same month and weekday as '||
       b.ename as msg
  from emp a, emp b
where to_char(a.hiredate,'DMON') =
      to_char(b.hiredate,'DMON')
  and a.empno < b.empno
order by a.ename
```

MySQL

After self-joining table EMP, use the DATE_FORMAT function to format the HIREDATE into weekday and month for comparison:

```
select concat(a.ename,
       ' was hired on the same month and weekday as ',
       b.ename) msg
  from emp a, emp b
 where date_format(a.hiredate,'%w%M') =
       date_format(b.hiredate,'%w%M')
   and a.empno < b.empno
order by a.ename
```

SQL Server

After self-joining table EMP, use the DATENAME function to format the HIREDATE into weekday and month for comparison:

```
1 select a.ename +
2        ' was hired on the same month and weekday as '+
3        b.ename msg
4  from emp a, emp b
5 where datename(dw,a.hiredate) = datename(dw,b.hiredate)
6   and datename(m,a.hiredate) = datename(m,b.hiredate)
7   and a.empno < b.empno
8 order by a.ename
```

Discussion

The only difference between the solutions is the date function used to format the HIREDATE. We'll use the Oracle/PostgreSQL solution in this discussion (because it's the shortest to type out), but the explanation holds true for the other solutions as well.

The first step is to self-join EMP so that each employee has access to the other employees' HIREDATEs. Consider the results of the query shown here (filtered for SCOTT):

```
select a.ename as scott, a.hiredate as scott_hd,
       b.ename as other_emps, b.hiredate as other_hds
  from emp a, emp b
 where a.ename = 'SCOTT'
  and a.empno != b.empno

SCOTT      SCOTT_HD    OTHER_EMPS OTHER_HDS
---------- ----------- ---------- -----------
SCOTT      09-DEC-2002 SMITH      17-DEC-2000
SCOTT      09-DEC-2002 ALLEN      20-FEB-2001
SCOTT      09-DEC-2002 WARD       22-FEB-2001
SCOTT      09-DEC-2002 JONES      02-APR-2001
SCOTT      09-DEC-2002 MARTIN     28-SEP-2001
SCOTT      09-DEC-2002 BLAKE      01-MAY-2001
SCOTT      09-DEC-2002 CLARK      09-JUN-2001
SCOTT      09-DEC-2002 KING       17-NOV-2001
SCOTT      09-DEC-2002 TURNER     08-SEP-2001
SCOTT      09-DEC-2002 ADAMS      12-JAN-2003
SCOTT      09-DEC-2002 JAMES      03-DEC-2001
SCOTT      09-DEC-2002 FORD       03-DEC-2001
SCOTT      09-DEC-2002 MILLER     23-JAN-2002
```

By self-joining table EMP, you can compare SCOTT's HIREDATE to the HIREDATE of all the other employees. The filter on EMPNO is so that SCOTT's HIREDATE is not returned as one of the OTHER_HDS. The next step is to use your RDBMS's

supplied date formatting function(s) to compare the weekday and month of the HIREDATEs and keep only those that match:

```
select a.ename as emp1, a.hiredate as emp1_hd,
       b.ename as emp2, b.hiredate as emp2_hd
  from emp a, emp b
 where to_char(a.hiredate,'DMON') =
       to_char(b.hiredate,'DMON')
   and a.empno != b.empno
 order by 1

EMP1       EMP1_HD     EMP2       EMP2_HD
---------- ----------- ---------- -----------
FORD       03-DEC-2001 SCOTT      09-DEC-2002
FORD       03-DEC-2001 JAMES      03-DEC-2001
JAMES      03-DEC-2001 SCOTT      09-DEC-2002
JAMES      03-DEC-2001 FORD       03-DEC-2001

SCOTT      09-DEC-2002 JAMES      03-DEC-2001
SCOTT      09-DEC-2002 FORD       03-DEC-2001
```

At this point, the HIREDATEs are correctly matched, but there are six rows in the result set rather than the three in the "Problem" section of this recipe. The reason for the extra rows is the filter on EMPNO. By using "not equals," you do not filter out the reciprocals. For example, the first row matches FORD and SCOTT, and the last row matches SCOTT and FORD. The six rows in the result set are technically accurate but redundant. To remove the redundancy, use "less than" (the HIREDATEs are removed to bring the intermediate queries closer to the final result set):

```
select a.ename as emp1, b.ename as emp2
  from emp a, emp b
 where to_char(a.hiredate,'DMON') =
       to_char(b.hiredate,'DMON')
   and a.empno < b.empno
 order by 1

EMP1       EMP2
---------- ----------
JAMES      FORD
SCOTT      JAMES
SCOTT      FORD
```

The final step is to simply concatenate the result set to form the message.

9.13 Identifying Overlapping Date Ranges

Problem

You want to find all instances of an employee starting a new project before ending an existing project. Consider table EMP_PROJECT:

```
select *
  from emp_project

EMPNO ENAME      PROJ_ID PROJ_START  PROJ_END
----- ---------- ------- ----------- -----------
7782  CLARK            1 16-JUN-2005 18-JUN-2005
7782  CLARK            4 19-JUN-2005 24-JUN-2005
7782  CLARK            7 22-JUN-2005 25-JUN-2005
7782  CLARK           10 25-JUN-2005 28-JUN-2005
7782  CLARK           13 28-JUN-2005 02-JUL-2005
7839  KING             2 17-JUN-2005 21-JUN-2005
7839  KING             8 23-JUN-2005 25-JUN-2005
7839  KING            14 29-JUN-2005 30-JUN-2005
7839  KING            11 26-JUN-2005 27-JUN-2005
7839  KING             5 20-JUN-2005 24-JUN-2005
7934  MILLER           3 18-JUN-2005 22-JUN-2005
7934  MILLER          12 27-JUN-2005 28-JUN-2005
7934  MILLER          15 30-JUN-2005 03-JUL-2005
7934  MILLER           9 24-JUN-2005 27-JUN-2005
7934  MILLER           6 21-JUN-2005 23-JUN-2005
```

Looking at the results for employee KING, you see that KING began PROJ_ID 8 before finishing PROJ_ID 5 and began PROJ_ID 5 before finishing PROJ_ID 2. You want to return the following result set:

```
EMPNO ENAME      MSG
----- ---------- --------------------------------
7782  CLARK      project 7 overlaps project 4
7782  CLARK      project 10 overlaps project 7
7782  CLARK      project 13 overlaps project 10
7839  KING       project 8 overlaps project 5
7839  KING       project 5 overlaps project 2
7934  MILLER     project 12 overlaps project 9
7934  MILLER     project 6 overlaps project 3
```

Solution

The key here is to find rows where PROJ_START (the date the new project starts) occurs on or after another project's PROJ_START date and on or before that other project's PROJ_END date. To begin, you need to be able to compare each project with each other project (for the same employee). By self-joining EMP_PROJECT on employee, you generate every possible combination of two projects for each employee. To find the overlaps, simply find the rows where PROJ_START for any PROJ_ID falls between PROJ_START and PROJ_END for another PROJ_ID by the same employee.

DB2, PostgreSQL, and Oracle

Self-join EMP_PROJECT. Then use the concatenation operator || to construct the message that explains which projects overlap:

```
1 select a.empno,a.ename,
2        'project '||b.proj_id||
3        ' overlaps project '||a.proj_id as msg
4   from emp_project a,
5        emp_project b
6  where a.empno = b.empno
7    and b.proj_start >= a.proj_start
8    and b.proj_start <= a.proj_end
9    and a.proj_id != b.proj_id
```

MySQL

Self-join EMP_PROJECT. Then use the CONCAT function to construct the message that explains which projects overlap:

```
1 select a.empno,a.ename,
2        concat('project ',b.proj_id,
3         ' overlaps project ',a.proj_id) as msg
4   from emp_project a,
5        emp_project b
6  where a.empno = b.empno
7    and b.proj_start >= a.proj_start
8    and b.proj_start <= a.proj_end
9    and a.proj_id != b.proj_id
```

SQL Server

Self-join EMP_PROJECT. Then use the concatenation operator + to construct the message that explains which projects overlap:

```
1 select a.empno,a.ename,
2        'project '+b.proj_id+
3        ' overlaps project '+a.proj_id as msg
4   from emp_project a,
5        emp_project b
6  where a.empno = b.empno
7    and b.proj_start >= a.proj_start
8    and b.proj_start <= a.proj_end
9    and a.proj_id != b.proj_id
```

Discussion

The only difference between the solutions lies in the string concatenation, so one discussion using the DB2 syntax will cover all three solutions. The first step is a self-join of EMP_PROJECT so that the PROJ_START dates can be compared among the different projects. The output of the self-join for employee KING is shown here. You can observe how each project can "see" the other projects:

```
select a.ename,
       a.proj_id as a_id,
       a.proj_start as a_start,
       a.proj_end as a_end,
       b.proj_id as b_id,
       b.proj_start as b_start
  from emp_project a,
       emp_project b
 where a.ename = 'KING'
   and a.empno = b.empno
   and a.proj_id != b.proj_id
order by 2

ENAME  A_ID  A_START     A_END       B_ID  B_START
------ ----- ----------- ----------- ----- -----------
KING       2 17-JUN-2005 21-JUN-2005     8 23-JUN-2005
KING       2 17-JUN-2005 21-JUN-2005    14 29-JUN-2005
KING       2 17-JUN-2005 21-JUN-2005    11 26-JUN-2005
KING       2 17-JUN-2005 21-JUN-2005     5 20-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005     2 17-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005     8 23-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005    11 26-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005    14 29-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005     2 17-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005    14 29-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005     5 20-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005    11 26-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005     2 17-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005     8 23-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005    14 29-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005     5 20-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005     2 17-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005     8 23-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005     5 20-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005    11 26-JUN-2005
```

As you can see from the result set, the self-join makes finding overlapping dates easy: simply return each row where B_START occurs between A_START and A_END. If you look at the WHERE clause on lines 7 and 8 of the solution:

```
and b.proj_start >= a.proj_start
and b.proj_start <= a.proj_end
```

it is doing just that. Once you have the required rows, constructing the messages is just a matter of concatenating the return values.

Oracle users can use the window function LEAD OVER to avoid the self-join, if the maximum number of projects per employee is fixed. This can come in handy if the self-join is expensive for your particular results (if the self-join requires more resources than the sorts needed for LEAD OVER). For example, consider the alternative for employee KING using LEAD OVER:

```
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,2)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,3)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,4)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       end is_overlap
  from emp_project
 where ename = 'KING'

EMPNO ENAME  PROJ_ID PROJ_START  PROJ_END    IS_OVERLAP
----- ------ ------- ----------- ----------- ----------
7839  KING         2 17-JUN-2005 21-JUN-2005          5
7839  KING         5 20-JUN-2005 24-JUN-2005          8
7839  KING         8 23-JUN-2005 25-JUN-2005
7839  KING        11 26-JUN-2005 27-JUN-2005
7839  KING        14 29-JUN-2005 30-JUN-2005
```

Because the number of projects is fixed at five for employee KING, you can use LEAD OVER to examine the dates of all the projects without a self-join. From here, producing the final result set is easy. Simply keep the rows where IS_OVERLAP is not NULL:

```
select empno,ename,
       'project '||is_overlap||
       ' overlaps project '||proj_id msg
  from (
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,2)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,3)over(order by proj_start)
```

```
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,4)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       end is_overlap
  from emp_project
 where ename = 'KING'
       )
 where is_overlap is not null

EMPNO ENAME  MSG
----- ------ --------------------------------
7839  KING   project 5 overlaps project 2
7839  KING   project 8 overlaps project 5
```

To allow the solution to work for all employees (not just KING), partition by ENAME in the LEAD OVER function:

```
select empno,ename,
       'project '||is_overlap||
       ' overlaps project '||proj_id msg
  from (
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,2)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,3)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,4)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       end is_overlap
 from emp_project
      )
where is_overlap is not null
```

```
EMPNO ENAME  MSG
----- ------ ------------------------------
7782  CLARK  project 7 overlaps project 4
7782  CLARK  project 10 overlaps project 7
7782  CLARK  project 13 overlaps project 10
7839  KING   project 5 overlaps project 2
7839  KING   project 8 overlaps project 5
7934  MILLER project 6 overlaps project 3
7934  MILLER project 12 overlaps project 9
```

9.14 Summing Up

Date manipulations are a common problem for anyone querying a database—a series of events stored with their dates inspires business users to ask creative date-based questions. At the same time, dates are one of the less standardized areas of SQLs between vendors. We hope that you take away from this chapter an idea of how even when the syntax is different, there is still a common logic that can be applied to queries that use dates.

CHAPTER 10

Working with Ranges

This chapter is about "everyday" queries that involve ranges. Ranges are common in everyday life. For example, projects that we work on range over consecutive periods of time. In SQL, it's often necessary to search for ranges, or to generate ranges, or to otherwise manipulate range-based data. The queries you'll read about here are slightly more involved than the queries found in the preceding chapters, but they are just as common, and they'll begin to give you a sense of what SQL can really do for you when you learn to take full advantage of it.

10.1 Locating a Range of Consecutive Values

Problem

You want to determine which rows represent a range of consecutive projects. Consider the following result set from view V, which contains data about a project and its start and end dates:

```
select *
  from V

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2020 02-JAN-2020
      2 02-JAN-2020 03-JAN-2020
      3 03-JAN-2020 04-JAN-2020
      4 04-JAN-2020 05-JAN-2020
      5 06-JAN-2020 07-JAN-2020
      6 16-JAN-2020 17-JAN-2020
      7 17-JAN-2020 18-JAN-2020
      8 18-JAN-2020 19-JAN-2020
      9 19-JAN-2020 20-JAN-2020
     10 21-JAN-2020 22-JAN-2020
```

```
        11 26-JAN-2020 27-JAN-2020
        12 27-JAN-2020 28-JAN-2020
        13 28-JAN-2020 29-JAN-2020
        14 29-JAN-2020 30-JAN-2020
```

Excluding the first row, each row's PROJ_START should equal the PROJ_END of the row before it ("before" is defined as PROJ_ID–1 for the current row). Examining the first five rows from view V, PROJ_IDs 1 through 3 are part of the same "group" as each PROJ_END equals the PROJ_START of the row after it. Because you want to find the range of dates for consecutive projects, you would like to return all rows where the current PROJ_END equals the next row's PROJ_START. If the first five rows comprised the entire result set, you would like to return only the first three rows. The final result set (using all 14 rows from view V) should be:

```
PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1  01-JAN-2020 02-JAN-2020
      2  02-JAN-2020 03-JAN-2020
      3  03-JAN-2020 04-JAN-2020
      6  16-JAN-2020 17-JAN-2020
      7  17-JAN-2020 18-JAN-2020
      8  18-JAN-2020 19-JAN-2020
     11  26-JAN-2020 27-JAN-2020
     12  27-JAN-2020 28-JAN-2020
     13  28-JAN-2020 29-JAN-2020
```

The rows with PROJ_IDs 4, 5, 9, 10, and 14 are excluded from this result set because the PROJ_END of each of these rows does not match the PROJ_START of the row following it.

Solution

This solution takes best advantage of the window function LEAD OVER to look at the "next" row's BEGIN_DATE, thus avoiding the need to self-join, which was necessary before window functions were widely introduced:

```
1 select proj_id, proj_start, proj_end
2   from (
3 select proj_id, proj_start, proj_end,
4        lead(proj_start)over(order by proj_id) next_proj_start
5   from V
6        ) alias
7 where next_proj_start = proj_end
```

Discussion

DB2, MySQL, PostgreSQL, SQL Server, and Oracle

Although it is possible to develop a solution using a self-join, the window function LEAD OVER is perfect for this type of problem, and more intuitive. The function LEAD OVER allows you to examine other rows without performing a self-join (though the function must impose order on the result set to do so). Consider the results of the inline view (lines 3–5) for IDs 1 and 4:

```
select *
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_proj_start
  from v
       )
 where proj_id in ( 1, 4 )

PROJ_ID PROJ_START  PROJ_END    NEXT_PROJ_START
------- ----------- ----------- ---------------
      1 01-JAN-2020 02-JAN-2020 02-JAN-2020
      4 04-JAN-2020 05-JAN-2020 06-JAN-2020
```

Examining this snippet of code and its result set, it is particularly easy to see why PROJ_ID 4 is excluded from the final result set of the complete solution. It's excluded because its PROJ_END date of 05-JAN-2020 does not match the "next" project's start date of 06-JAN-2020.

The function LEAD OVER is extremely handy when it comes to problems such as this one, particularly when examining partial results. When working with window functions, keep in mind that they are evaluated after the FROM and WHERE clauses, so the LEAD OVER function in the preceding query must be embedded within an inline view. Otherwise, the LEAD OVER function is applied to the result set after the WHERE clause has filtered out all rows except for PROJ_ID's 1 and 4.

Now, depending on how you view the data, you may very well want to include PROJ_ID 4 in the final result set. Consider the first five rows from view V:

```
select *
  from V
 where proj_id <= 5

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2020 02-JAN-2020
      2 02-JAN-2020 03-JAN-2020
      3 03-JAN-2020 04-JAN-2020
      4 04-JAN-2020 05-JAN-2020
      5 06-JAN-2020 07-JAN-2020
```

If your requirement is such that PROJ_ID 4 is in fact contiguous (because PROJ_START for PROJ_ID 4 matches PROJ_END for PROJ_ID 3), and that only PROJ_ ID 5 should be discarded, the proposed solution for this recipe is incorrect (!) or, at the very least, incomplete:

```
select proj_id, proj_start, proj_end
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_start
  from V
where proj_id <= 5
      )
where proj_end = next_start

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2020 02-JAN-2020
      2 02-JAN-2020 03-JAN-2020
      3 03-JAN-2020 04-JAN-2020
```

If you believe PROJ_ID 4 should be included, simply add LAG OVER to the query and use an additional filter in the WHERE clause:

```
select proj_id, proj_start, proj_end
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_start,
       lag(proj_end)over(order by proj_id) last_end
  from V
where proj_id <= 5
      )
where proj_end = next_start
   or proj_start = last_end

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2020 02-JAN-2020
      2 02-JAN-2020 03-JAN-2020
      3 03-JAN-2020 04-JAN-2020
      4 04-JAN-2020 05-JAN-2020
```

Now PROJ_ID 4 is included in the final result set, and only the evil PROJ_ID 5 is excluded. Please consider your exact requirements when applying these recipes to your code.

10.2 Finding Differences Between Rows in the Same Group or Partition

Problem

You want to return the DEPTNO, ENAME, and SAL of each employee along with the difference in SAL between employees in the same department (i.e., having the same value for DEPTNO). The difference should be between each current employee and the employee hired immediately afterward (you want to see if there is a correlation between seniority and salary on a "per department" basis). For each employee hired last in his department, return "N/A" for the difference. The result set should look like this:

```
DEPTNO ENAME             SAL HIREDATE    DIFF
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-2006      -2550
    10 KING             5000 17-NOV-2006       3700
    10 MILLER           1300 23-JAN-2007        N/A
    20 SMITH             800 17-DEC-2005      -2175
    20 JONES            2975 02-APR-2006        -25
    20 FORD             3000 03-DEC-2006          0
    20 SCOTT            3000 09-DEC-2007       1900
    20 ADAMS            1100 12-JAN-2008        N/A
    30 ALLEN            1600 20-FEB-2006        350
    30 WARD             1250 22-FEB-2006      -1600
    30 BLAKE            2850 01-MAY-2006       1350
    30 TURNER           1500 08-SEP-2006        250
    30 MARTIN           1250 28-SEP-2006        300
    30 JAMES             950 03-DEC-2006        N/A
```

Solution

The is another example of where the window functions LEAD OVER and LAG OVER come in handy. You can easily access next and prior rows without additional joins. Alternative methods such as subqueries or self-joins are possible but awkward:

```
with next_sal_tab (deptno,ename,sal,hiredate,next_sal)
as
(select deptno, ename, sal, hiredate,
      lead(sal)over(partition by deptno
                        order by hiredate) as next_sal
 from emp )

   select deptno, ename, sal, hiredate
,    coalesce(cast(sal-next_sal as char), 'N/A') as diff
   from next_sal_tab
```

In this case, for the sake of variety, we have used a CTE rather than a subquery—both will work across most RDBMSs these days, with the preference usually relating to readability.

Discussion

The first step is to use the LEAD OVER window function to find the "next" salary for each employee within their department. The employees hired last in each department will have a NULL value for NEXT_SAL:

```
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) as next_sal
  from emp

DEPTNO ENAME             SAL HIREDATE      NEXT_SAL
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-2006       5000
    10 KING             5000 17-NOV-2006       1300
    10 MILLER           1300 23-JAN-2007
    20 SMITH             800 17-DEC-2005       2975
    20 JONES            2975 02-APR-2006       3000
    20 FORD             3000 03-DEC-2006       3000
    20 SCOTT            3000 09-DEC-2007       1100
    20 ADAMS            1100 12-JAN-2008
    30 ALLEN            1600 20-FEB-2006       1250
    30 WARD             1250 22-FEB-2006       2850
    30 BLAKE            2850 01-MAY-2006       1500
    30 TURNER           1500 08-SEP-2006       1250
    30 MARTIN           1250 28-SEP-2006        950
    30 JAMES             950 03-DEC-2006
```

The next step is to take the difference between each employee's salary and the salary of the employee hired immediately after them in the same department:

```
select deptno,ename,sal,hiredate, sal-next_sal diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) next_sal
  from emp
       )

DEPTNO ENAME             SAL HIREDATE          DIFF
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-2006      -2550
    10 KING             5000 17-NOV-2006       3700
    10 MILLER           1300 23-JAN-2007
    20 SMITH             800 17-DEC-2005      -2175
    20 JONES            2975 02-APR-2006        -25
    20 FORD             3000 03-DEC-2006          0
    20 SCOTT            3000 09-DEC-2007       1900
    20 ADAMS            1100 12-JAN-2008
    30 ALLEN            1600 20-FEB-2006        350
```

```
    30 WARD             1250 22-FEB-2006      -1600
    30 BLAKE            2850 01-MAY-2006       1350
    30 TURNER           1500 08-SEP-2006        250
    30 MARTIN           1250 28-SEP-2006        300
    30 JAMES             950 03-DEC-2006
```

The next step is to use the COALESCE function to insert "N/A" when there is no next salary. To be able to return "N/A" you must cast the value of DIFF to a string:

```
select deptno,ename,sal,hiredate,
       nvl(to_char(sal-next_sal),'N/A') diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) next_sal
  from emp
       )

DEPTNO ENAME             SAL HIREDATE    DIFF
------ ---------- ---------- ----------- ---------------
    10 CLARK            2450 09-JUN-2006 -2550
    10 KING             5000 17-NOV-2006 3700
    10 MILLER           1300 23-JAN-2007 N/A
    20 SMITH             800 17-DEC-2005 -2175
    20 JONES            2975 02-APR-2006 -25
    20 FORD             3000 03-DEC-2006 0
    20 SCOTT            3000 09-DEC-2007 1900
    20 ADAMS            1100 12-JAN-2008 N/A
    30 ALLEN            1600 20-FEB-2006 350
    30 WARD             1250 22-FEB-2006 -1600
    30 BLAKE            2850 01-MAY-2006 1350
    30 TURNER           1500 08-SEP-2006 250
    30 MARTIN           1250 28-SEP-2006 300
    30 JAMES             950 03-DEC-2006 N/A
```

While the majority of the solutions provided in this book do not deal with "what if" scenarios (for the sake of readability and the author's sanity), the scenario involving duplicates when using the LEAD OVER function in this manner must be discussed. In the simple sample data in table EMP, no employees have duplicate HIREDATEs, yet this is an unlikely situation. Normally, we would not discuss a "what if" situation such as duplicates (since there aren't any in table EMP), but the workaround involving LEAD may not be immediately obvious. Consider the following query, which returns the difference in SAL between the employees in DEPTNO 10 (the difference is performed in the order in which they were hired):

```
select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                     order by hiredate) next_sal
  from emp
```

```
 where deptno=10 and empno > 10
       )

DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-2006      -2550
    10 KING    5000 17-NOV-2006       3700
    10 MILLER  1300 23-JAN-2007        N/A
```

This solution is correct considering the data in table EMP, but if there were duplicate rows, the solution would fail. Consider the following example, which shows four more employees hired on the same day as KING:

```
insert into emp (empno,ename,deptno,sal,hiredate)
values (1,'ant',10,1000,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (2,'joe',10,1500,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (3,'jim',10,1600,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (4,'jon',10,1700,to_date('17-NOV-2006'))

select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                        order by hiredate) next_sal
  from emp
 where deptno=10
       )

DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-2006       1450
    10 ant     1000 17-NOV-2006       -500
    10 joe     1500 17-NOV-2006      -3500
    10 KING    5000 17-NOV-2006       3400
    10 jim     1600 17-NOV-2006       -100
    10 jon     1700 17-NOV-2006        400
    10 MILLER  1300 23-JAN-2007        N/A
```

You'll notice that with the exception of employee JON, all employees hired on the same date (November 17) evaluate their salary against another employee hired on the same date! This is incorrect. All employees hired on November 17 should have the difference of salary computed against MILLER's salary, not another employee hired on November 17. Take, for example, employee ANT. The value for DIFF for ANT is –500 because ANT's SAL is compared with JOE's SAL and is 500 less than JOE's

SAL, hence the value of –500. The correct value for DIFF for employee ANT should be –300 because ANT makes 300 less than MILLER, who is the next employee hired by HIREDATE. The reason the solution seems to not work is due to the default behavior of Oracle's LEAD OVER function. By default, LEAD OVER looks ahead only one row. So, for employee ANT, the next SAL based on HIREDATE is JOE's SAL, because LEAD OVER simply looks one row ahead and doesn't skip duplicates. Fortunately, Oracle planned for such a situation and allows you to pass an additional parameter to LEAD OVER to determine how far ahead it should look. In the previous example, the solution is simply a matter of counting: find the distance from each employee hired on November 17 to January 23 (MILLER's HIREDATE). The following shows how to accomplish this:

```
select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal,cnt-rn+1)over(partition by deptno
                          order by hiredate) next_sal
  from (
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate order by sal) rn
  from emp
 where deptno=10
       )
       )

DEPTNO ENAME     SAL HIREDATE    DIFF
------ ------  ----- ----------- ----------
    10 CLARK    2450 09-JUN-2006       1450
    10 ant      1000 17-NOV-2006       -300
    10 joe      1500 17-NOV-2006        200
    10 jim      1600 17-NOV-2006        300
    10 jon      1700 17-NOV-2006        400
    10 KING     5000 17-NOV-2006       3700
    10 MILLER   1300 23-JAN-2007        N/A
```

Now the solution is correct. As you can see, all the employees hired on November 17 now have their salaries compared with MILLER's salary. Inspecting the results, employee ANT now has a value of –300 for DIFF, which is what we were hoping for. If it isn't immediately obvious, the expression passed to LEAD OVER; CNT-RN+1 is simply the distance from each employee hired on November 17 to MILLER. Consider the following inline view, which shows the values for CNT and RN:

```
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate order by sal) rn
  from emp
 where deptno=10
```

```
DEPTNO ENAME    SAL HIREDATE           CNT         RN
------ ------ ----- ----------- ---------- ----------
    10 CLARK   2450 09-JUN-2006          1          1
    10 ant     1000 17-NOV-2006          5          1
    10 joe     1500 17-NOV-2006          5          2
    10 jim     1600 17-NOV-2006          5          3
    10 jon     1700 17-NOV-2006          5          4
    10 KING    5000 17-NOV-2006          5          5
    10 MILLER  1300 23-JAN-2007          1          1
```

The value for CNT represents, for each employee with a duplicate HIREDATE, how many duplicates there are in total for their HIREDATE. The value for RN represents a ranking for the employees in DEPTNO 10. The rank is partitioned by DEPTNO and HIREDATE so only employees with a HIREDATE that another employee has will have a value greater than one. The ranking is sorted by SAL (this is arbitrary; SAL is convenient, but we could have just as easily chosen EMPNO). Now that you know how many total duplicates there are and you have a ranking of each duplicate, the distance to MILLER is simply the total number of duplicates minus the current rank plus one (CNT-RN+1). The results of the distance calculation and its effect on LEAD OVER are shown here:

```
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                         order by hiredate) incorrect,
       cnt-rn+1 distance,
       lead(sal,cnt-rn+1)over(partition by deptno
                         order by hiredate) correct
  from (
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate
                           order by sal) rn
  from emp
 where deptno=10
       )

DEPTNO ENAME    SAL HIREDATE    INCORRECT    DISTANCE    CORRECT
------ ------ ----- ----------- ---------- ---------- ----------
    10 CLARK   2450 09-JUN-2006       1000          1       1000
    10 ant     1000 17-NOV-2006       1500          5       1300
    10 joe     1500 17-NOV-2006       1600          4       1300
    10 jim     1600 17-NOV-2006       1700          3       1300
    10 jon     1700 17-NOV-2006       5000          2       1300
    10 KING    5000 17-NOV-2006       1300          1       1300
    10 MILLER  1300 23-JAN-2007                     1
```

Now you can clearly see the effect that you have when you pass the correct distance to LEAD OVER. The rows for INCORRECT represent the values returned by LEAD OVER using a default distance of one. The rows for CORRECT represent the values returned by LEAD OVER using the proper distance for each employee with a

duplicate HIREDATE to MILLER. At this point, all that is left is to find the difference between CORRECT and SAL for each row, which has already been shown.

10.3 Locating the Beginning and End of a Range of Consecutive Values

Problem

This recipe is an extension of the prior recipe, and it uses the same view V from the prior recipe. Now that you've located the ranges of consecutive values, you want to find just their start and end points. Unlike the prior recipe, if a row is not part of a set of consecutive values, you still want to return it. Why? Because such a row represents both the beginning and end of its range. Using the data from view V:

```
select *
  from V


PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2020 02-JAN-2020
      2 02-JAN-2020 03-JAN-2020
      3 03-JAN-2020 04-JAN-2020
      4 04-JAN-2020 05-JAN-2020
      5 06-JAN-2020 07-JAN-2020
      6 16-JAN-2020 17-JAN-2020
      7 17-JAN-2020 18-JAN-2020
      8 18-JAN-2020 19-JAN-2020
      9 19-JAN-2020 20-JAN-2020
     10 21-JAN-2020 22-JAN-2020
     11 26-JAN-2020 27-JAN-2020
     12 27-JAN-2020 28-JAN-2020
     13 28-JAN-2020 29-JAN-2020
     14 29-JAN-2020 30-JAN-2020
```

you want the final result set to be as follows:

```
PROJ_GRP PROJ_START  PROJ_END
-------- ----------- -----------
       1 01-JAN-2020 05-JAN-2020
       2 06-JAN-2020 07-JAN-2020
       3 16-JAN-2020 20-JAN-2020
       4 21-JAN-2020 22-JAN-2020
       5 26-JAN-2020 30-JAN-2020
```

Solution

This problem is a bit more involved than its predecessor. First, you must identify what the ranges are. A range of rows is defined by the values for PROJ_START and PROJ_END. For a row to be considered "consecutive" or part of a group, its PROJ_START value must equal the PROJ_END value of the row before it. In the case where a row's PROJ_START value does not equal the prior row's PROJ_END value and its PROJ_END value does not equal the next row's PROJ_START value, this is an instance of a single row group. Once you have identify the ranges, you need to be able to group the rows in these ranges together (into groups) and return only their start and end points.

Examine the first row of the desired result set. The PROJ_START is the PROJ_START for PROJ_ID 1 from view V, and the PROJ_END is the PROJ_END for PROJ_ID 4 from view V. Despite the fact that PROJ_ID 4 does not have a consecutive value following it, it is the last of a range of consecutive values, and thus it is included in the first group.

The most straightforward approach for this problem is to use the LAG OVER window function. Use LAG OVER to determine whether each prior row's PROJ_END equals the current row's PROJ_START to help place the rows into groups. Once they are grouped, use the aggregate functions MIN and MAX to find their start and end points:

```
 1 select proj_grp, min(proj_start), max(proj_end)
 2   from (
 3 select proj_id,proj_start,proj_end,
 4        sum(flag)over(order by proj_id) proj_grp
 5   from (
 6 select proj_id,proj_start,proj_end,
 7        case when
 8             lag(proj_end)over(order by proj_id) = proj_start
 9             then 0 else 1
10        end flag
11   from V
12        ) alias1
13        ) alias2
14  group by proj_grp
```

Discussion

The window function LAG OVER is extremely useful in this situation. You can examine each prior row's PROJ_END value without a self-join, without a scalar subquery, and without a view. The results of the LAG OVER function without the CASE expression are as follows:

```
select proj_id,proj_start,proj_end,
      lag(proj_end)over(order by proj_id) prior_proj_end
  from V


PROJ_ID PROJ_START  PROJ_END    PRIOR_PROJ_END
------- ----------- ----------- --------------
      1 01-JAN-2020 02-JAN-2020
      2 02-JAN-2020 03-JAN-2020 02-JAN-2020
      3 03-JAN-2020 04-JAN-2020 03-JAN-2020
      4 04-JAN-2020 05-JAN-2020 04-JAN-2020
      5 06-JAN-2020 07-JAN-2020 05-JAN-2020
      6 16-JAN-2020 17-JAN-2020 07-JAN-2020
      7 17-JAN-2020 18-JAN-2020 17-JAN-2020
      8 18-JAN-2020 19-JAN-2020 18-JAN-2020
      9 19-JAN-2020 20-JAN-2020 19-JAN-2020
     10 21-JAN-2020 22-JAN-2020 20-JAN-2020
     11 26-JAN-2020 27-JAN-2020 22-JAN-2020
     12 27-JAN-2020 28-JAN-2020 27-JAN-2020
     13 28-JAN-2020 29-JAN-2020 28-JAN-2020
     14 29-JAN-2020 30-JAN-2020 29-JAN-2020
```

The CASE expression in the complete solution simply compares the value returned by LAG OVER to the current row's PROJ_START value; if they are the same, return 0, else return 1. The next step is to create a running total on the zeros and ones returned by the CASE expression to put each row into a group. The results of the running total are shown here:

```
select proj_id,proj_start,proj_end,
       sum(flag)over(order by proj_id) proj_grp
  from (
select proj_id,proj_start,proj_end,
       case when
            lag(proj_end)over(order by proj_id) = proj_start
            then 0 else 1
       end flag
  from V
       )


PROJ_ID PROJ_START  PROJ_END      PROJ_GRP
------- ----------- ----------- ----------
      1 01-JAN-2020 02-JAN-2020          1
      2 02-JAN-2020 03-JAN-2020          1
      3 03-JAN-2020 04-JAN-2020          1
      4 04-JAN-2020 05-JAN-2020          1
      5 06-JAN-2020 07-JAN-2020          2
      6 16-JAN-2020 17-JAN-2020          3
      7 17-JAN-2020 18-JAN-2020          3
      8 18-JAN-2020 19-JAN-2020          3
      9 19-JAN-2020 20-JAN-2020          3
     10 21-JAN-2020 22-JAN-2020          4
```

```
11 26-JAN-2020 27-JAN-2020          5
12 27-JAN-2020 28-JAN-2020          5
13 28-JAN-2020 29-JAN-2020          5
14 29-JAN-2020 30-JAN-2020          5
```

Now that each row has been placed into a group, simply use the aggregate functions MIN and MAX on PROJ_START and PROJ_END, respectively, and group by the values created in the PROJ_GRP running total column.

10.4 Filling in Missing Values in a Range of Values

Problem

You want to return the number of employees hired each year for the entire decade of the 2005s, but there are some years in which no employees were hired. You would like to return the following result set:

```
YR          CNT
---- ----------
2005          1
2006         10
2007          2
2008          1
2009          0
2010          0
2011          0
2012          0
2013          0
2014          0
```

Solution

The trick to this solution is returning zeros for years that saw no employees hired. If no employee was hired in a given year, then no rows for that year will exist in table EMP. If the year does not exist in the table, how can you return a count, any count, even zero? The solution requires you to outer join. You must supply a result set that returns all the years you want to see, and then perform a count against table EMP to see if there were any employees hired in each of those years.

DB2

Use table EMP as a pivot table (because it has 14 rows) and the built-in function YEAR to generate one row for each year in the decade of 2005. Outer join to table EMP and count how many employees were hired each year:

```
1 select x.yr, coalesce(y.cnt,0) cnt
2   from (
3 select year(min(hiredate)over()) -
4        mod(year(min(hiredate)over()),10) +
```

```
 5        row_number()over()-1 yr
 6   from emp fetch first 10 rows only
 7        ) x
 8   left join
 9        (
10 select year(hiredate) yr1, count(*) cnt
11   from emp
12  group by year(hiredate)
13        ) y
14     on ( x.yr = y.yr1 )
```

Oracle

The Oracle solution follows the same structure as the DB2 solution, with only the differences in the syntax Oracle handles causing a distinct solution to be required:

```
 1 select x.yr, coalesce(cnt,0) cnt
 2   from (
 3 select extract(year from min(hiredate)over()) -
 4        mod(extract(year from min(hiredate)over()),10) +
 5        rownum-1 yr
 6   from emp
 7  where rownum <= 10
 8        ) x
 9   left join
10        (
11 select to_number(to_char(hiredate,'YYYY')) yr, count(*) cnt
12   from emp
13  group by to_number(to_char(hiredate,'YYYY'))
14        ) y
15     on ( x.yr = y.yr )
```

PostgreSQL and MySQL

Use table T10 as a pivot table (because it has 10 rows) and the built-in function EXTRACT to generate one row for each year in the decade of 2005. Outer join to table EMP and count how many employees were hired each year:

```
 1 select y.yr, coalesce(x.cnt,0) as cnt
 2   from (
 3 selectmin_year-mod(cast(min_year as int),10)+rn as yr
 4   from (
 5 select (select min(extract(year from hiredate))
 6           from emp) as min_year,
 7        id-1 as rn
 8   from t10
 9        ) a
10        ) y
11   left join
12        (
13 select extract(year from hiredate) as yr, count(*) as cnt
14   from emp
```

```
 group by extract(year from hiredate)
       ) x
    on ( y.yr = x.yr )
```

SQL Server

Use table EMP as a pivot table (because it has 14 rows) and the built-in function YEAR to generate one row for each year in the decade of 2005. Outer join to table EMP and count how many employees were hired each year:

```
select x.yr, coalesce(y.cnt,0) cnt
  from (
select top (10)
       (year(min(hiredate)over()) -
        year(min(hiredate)over())%10)+
        row_number()over(order by hiredate)-1 yr
  from emp
       ) x
  left join
       (
select year(hiredate) yr, count(*) cnt
  from emp
 group by year(hiredate)
       ) y
    on ( x.yr = y.yr )
```

Discussion

Despite the difference in syntax, the approach is the same for all solutions. Inline view X returns each year in the decade of the '80s by first finding the year of the earliest HIREDATE. The next step is to add RN–1 to the difference between the earliest year and the earliest year modulus ten. To see how this works, simply execute inline view X and return each of the values involved separately. Listed here is the result set for inline view X using the window function MIN OVER (DB2, Oracle, SQL Server) and a scalar subquery (MySQL, PostgreSQL):

```
select year(min(hiredate)over()) -
       mod(year(min(hiredate)over()),10) +
       row_number()over()-1 yr,
       year(min(hiredate)over()) min_year,
       mod(year(min(hiredate)over()),10) mod_yr,
       row_number()over()-1 rn
  from emp fetch first 10 rows only

  YR   MIN_YEAR     MOD_YR         RN
---- ---------- ---------- ----------
2005       2005          0          0
2006       2005          0          1
2007       2005          0          2
2008       2005          0          3
1984       2005          0          4
```

```
2010       2005          0          5
2011       2005          0          6
2012       2005          0          7
2013       2005          0          8
2014       2005          0          9
```

```
select min_year-mod(min_year,10)+rn as yr,
       min_year,
       mod(min_year,10) as mod_yr
       rn
  from (
select (select min(extract(year from hiredate))
          from emp) as min_year,
        id-1 as rn
  from t10
       ) x

  YR   MIN_YEAR     MOD_YR         RN
---- ---------- ---------- ----------
2005       2005          0          0
2006       2005          0          1
2007       2005          0          2
2008       2005          0          3
2009       2005          0          4
2010       2005          0          5
2011       2005          0          6
2012       2005          0          7
2013       2005          0          8
2014       2005          0          9
```

Inline view Y returns the year for each HIREDATE and the number of employees hired during that year:

```
select year(hiredate) yr, count(*) cnt
  from emp
 group by year(hiredate)

   YR        CNT
----- ----------
 2005          1
 2006         10
 2007          2
 2008          1
```

Finally, outer join inline view Y to inline view X so that every year is returned even if there are no employees hired.

10.5 Generating Consecutive Numeric Values

Problem

You would like to have a "row source generator" available to you in your queries. Row source generators are useful for queries that require pivoting. For example, you want to return a result set such as the following, up to any number of rows that you specify:

```
ID
---
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
…
```

If your RDBMS provides built-in functions for returning rows dynamically, you do not need to create a pivot table in advance with a fixed number of rows. That's why a dynamic row generator can be so handy. Otherwise, you must use a traditional pivot table with a fixed number of rows (that may not always be enough) to generate rows when needed.

Solution

This solution shows how to return 10 rows of increasing numbers starting from 1. You can easily adapt the solution to return any number of rows.

The ability to return increasing values from one opens the door to many other solutions. For example, you can generate numbers to add to dates in order to generate sequences of days. You can also use such numbers to parse through strings.

DB2 and SQL Server

Use the recursive WITH clause to generate a sequence of rows with incrementing values. Using a recursive CTE will in fact work with the majority of RDBMSs today:

```
with x (id)
as (
select 1
 union all
select id+1
  from x
 where id+1 <= 10
```

```
 8 )
 9 select * from x
```

Oracle

In Oracle Database you can generate rows using the MODEL clause:

```
1 select array id
2   from dual
3  model
4    dimension by (0 idx)
5    measures(1 array)
6    rules iterate (10) (
7      array[iteration_number] = iteration_number+1
8    )
```

PostgreSQL

Use the handy function GENERATE_SERIES, which is designed for the express purpose of generating rows:

```
1 select id
2   from generate_series (1, 10) x(id)
```

Discussion

DB2 and SQL Server

The recursive WITH clause increments ID (which starts at one) until the WHERE clause is satisfied. To kick things off, you must generate one row having the value 1. You can do this by selecting 1 from a one-row table or, in the case of DB2, by using the VALUES clause to create a one-row result set.

Oracle

In the MODEL clause solution, there is an explicit ITERATE command that allows you to generate multiple rows. Without the ITERATE clause, only one row will be returned, since DUAL has only one row. For example:

```
select array id
  from dual
model
  dimension by (0 idx)
  measures(1 array)
  rules ()

 ID
 --
  1
```

The MODEL clause not only allows you array access to rows, it allows you to easily "create" or return rows that are not in the table you are selecting against. In this solution, IDX is the array index (location of a specific value in the array) and ARRAY (aliased ID) is the "array" of rows. The first row defaults to 1 and can be referenced with ARRAY[0]. Oracle provides the function ITERATION_NUMBER so you can track the number of times you've iterated. The solution iterates 10 times, causing ITERATION_NUMBER to go from 0 to 9. Adding one to each of those values yields the results 1 through 10.

It may be easier to visualize what's happening with the model clause if you execute the following query:

```
select 'array['||idx||'] = '||array as output
  from dual
 model
   dimension by (0 idx)
   measures(1 array)
   rules iterate (10) (
     array[iteration_number] = iteration_number+1
   )

OUTPUT
------------------
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
array[5] = 6
array[6] = 7
array[7] = 8
array[8] = 9
array[9] = 10
```

PostgreSQL

All the work is done by the function GENERATE_SERIES. The function accepts three parameters, all numeric values. The first parameter is the start value, the second parameter is the ending value, and the third parameter is an optional "step" value (how much each value is incremented by). If you do not pass a third parameter, the increment defaults to one.

The GENERATE_SERIES function is flexible enough so that you do not have to hardcode parameters. For example, if you wanted to return 5 rows starting from value 10 and ending with value 30, incrementing by 5 such that the result set is the following:

```
 ID
---
 10
 15
 20
 25
 30
```

you can be creative and do something like this:

```
select id
  from generate_series(
         (select min(deptno) from emp),
         (select max(deptno) from emp),
         5
       ) x(id)
```

Notice here that the actual values passed to GENERATE_SERIES are not known when the query is written. Instead, they are generated by subqueries when the main query executes.

10.6 Summing Up

Queries that take into account ranges are one of the most common requests from business users—they are a natural consquence of the way that businesses operate. At least some of the time, however, a degree of dexterity is needed to apply the range correctly, and the recipes in this chapter should demonstrate how to apply that dexterity.

CHAPTER 11

Advanced Searching

In a very real sense, this entire book so far has been about searching. You've seen all sorts of queries that use joins and WHERE clauses and grouping techniques to search out and return the results you need. Some types of searching operations, though, stand apart from others in that they represent a different way of thinking about searching. Perhaps you're displaying a result set one page at a time. Half of that problem is to identify (search for) the entire set of records that you want to display. The other half of that problem is to repeatedly search for the next page to display as a user cycles through the records on a display. Your first thought may not be to think of pagination as a searching problem, but it *can* be thought of that way, and it can be solved that way; that is the type of searching solution this chapter is all about.

11.1 Paginating Through a Result Set

Problem

You want to paginate or "scroll through" a result set. For example, you want to return the first five salaries from table EMP, then the next five, and so forth. Your goal is to allow a user to view five records at a time, scrolling forward with each click of a *Next* button.

Solution

Because there is no concept of first, last, or next in SQL, you must impose order on the rows you are working with. Only by imposing order can you accurately return ranges of records.

Use the window function ROW_NUMBER OVER to impose order, and specify the window of records that you want returned in your WHERE clause. For example, use this to return rows 1 through 5:

```
select sal
  from (
select row_number() over (order by sal) as rn,
       sal
  from emp
       ) x
 where rn between 1 and 5


 SAL
----
 800
 950
1100
1250
1250
```

Then use this to return rows 6 through 10:

```
select sal
  from (
select row_number() over (order by sal) as rn,
       sal
  from emp
       ) x
 where rn between 6 and 10


  SAL
-----
 1300
 1500
 1600
 2450
 2850
```

You can return any range of rows that you want simply by changing the WHERE clause of your query.

Discussion

The window function ROW_NUMBER OVER in inline view X will assign a unique number to each salary (in increasing order starting from 1). Listed here is the result set for inline view X:

```
select row_number() over (order by sal) as rn,
       sal
  from emp
```

```
RN         SAL
--  ----------
 1         800
 2         950
 3        1100
 4        1250
 5        1250
 6        1300
 7        1500
 8        1600
 9        2450
10        2850
11        2975
12        3000
13        3000
14        5000
```

Once a number has been assigned to a salary, simply pick the range you want to return by specifying values for RN.

For Oracle users, an alternative: you can use ROWNUM instead of ROW NUMBER OVER to generate sequence numbers for the rows:

```
select sal
  from (
select sal, rownum rn
  from (
select sal
  from emp
 order by sal
       )
       )
 where rn between 6 and 10

  SAL
-----
 1300
 1500
 1600
 2450
 2850
```

Using ROWNUM forces you into writing an extra level of subquery. The innermost subquery sorts rows by salary. The next outermost subquery applies row numbers to those rows, and, finally, the very outermost SELECT returns the data you are after.

11.2 Skipping n Rows from a Table

Problem

You want a query to return every other employee in table EMP; you want the first employee, third employee, and so forth. For example, from the following result set:

```
ENAME
--------
ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD
```

you want to return the following:

```
ENAME
----------
ADAMS
BLAKE
FORD
JONES
MARTIN
SCOTT
TURNER
```

Solution

To skip the second or fourth or *n*th row from a result set, you must impose order on the result set; otherwise, there is no concept of first or next, second, or fourth.

Use the window function ROW_NUMBER OVER to assign a number to each row, which you can then use in conjunction with the modulo function to skip unwanted rows. The modulo function is MOD for DB2, MySQL, PostgreSQL, and Oracle. In SQL Server, use the percent (%) operator. The following example uses MOD to skip even-numbered rows:

```
select ename
  from (
select row_number() over (order by ename) rn,
       ename
```

```
5    from emp
6         ) x
7   where mod(rn,2) = 1
```

Discussion

The call to the window function ROW_NUMBER OVER in inline view X will assign a rank to each row (no ties, even with duplicate names). The results are shown here:

```
select row_number() over (order by ename) rn, ename
  from emp

RN ENAME
-- --------
 1 ADAMS
 2 ALLEN
 3 BLAKE
 4 CLARK
 5 FORD
 6 JAMES
 7 JONES
 8 KING
 9 MARTIN
10 MILLER
11 SCOTT
12 SMITH
13 TURNER
14 WARD
```

The last step is to simply use modulus to skip every other row.

11.3 Incorporating OR Logic When Using Outer Joins

Problem

You want to return the name and department information for all employees in departments 10 and 20 along with department information for departments 30 and 40 (but no employee information). Your first attempt looks like this:

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d, emp e
 where d.deptno = e.deptno
   and (e.deptno = 10 or e.deptno = 20)
 order by 2

ENAME       DEPTNO DNAME          LOC
------- ---------- -------------- -----------
CLARK           10 ACCOUNTING     NEW YORK
KING            10 ACCOUNTING     NEW YORK
MILLER          10 ACCOUNTING     NEW YORK
SMITH           20 RESEARCH       DALLAS
```

```
ADAMS           20 RESEARCH       DALLAS
FORD            20 RESEARCH       DALLAS
SCOTT           20 RESEARCH       DALLAS
JONES           20 RESEARCH       DALLAS
```

Because the join in this query is an inner join, the result set does not include department information for DEPTNOs 30 and 40.

You attempt to outer join EMP to DEPT with the following query, but you still do not get the correct results:

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d left join emp e
    on (d.deptno = e.deptno)
 where e.deptno = 10
    or e.deptno = 20
 order by 2

ENAME       DEPTNO DNAME        LOC
------- ---------- ------------ -----------
CLARK           10 ACCOUNTING   NEW YORK
KING            10 ACCOUNTING   NEW YORK
MILLER          10 ACCOUNTING   NEW YORK
SMITH           20 RESEARCH     DALLAS
ADAMS           20 RESEARCH     DALLAS
FORD            20 RESEARCH     DALLAS
SCOTT           20 RESEARCH     DALLAS
JONES           20 RESEARCH     DALLAS
```

Ultimately, you would like the result set to be the following:

```
ENAME       DEPTNO DNAME        LOC
------- ---------- ------------ ---------
CLARK           10 ACCOUNTING    NEW YORK
KING            10 ACCOUNTING    NEW YORK
MILLER          10 ACCOUNTING    NEW YORK
SMITH           20 RESEARCH      DALLAS
JONES           20 RESEARCH      DALLAS
SCOTT           20 RESEARCH      DALLAS
ADAMS           20 RESEARCH      DALLAS
FORD            20 RESEARCH      DALLAS
                30 SALES         CHICAGO
                40 OPERATIONS    BOSTON
```

Solution

Move the OR condition into the JOIN clause:

```
1  select e.ename, d.deptno, d.dname, d.loc
2    from dept d left join emp e
3      on (d.deptno = e.deptno
4         and (e.deptno=10 or e.deptno=20))
5   order by 2
```

Alternatively, you can filter on EMP.DEPTNO first in an inline view and then outer join:

```
1  select e.ename, d.deptno, d.dname, d.loc
2    from dept d
3    left join
4         (select ename, deptno
5            from emp
6           where deptno in ( 10, 20 )
7         ) e on ( e.deptno = d.deptno )
8  order by 2
```

Discussion

DB2, MySQL, PostgreSQL, and SQL Server

Two solutions are given for these products. The first moves the OR condition into the JOIN clause, making it part of the join condition. By doing that, you can filter the rows returned from EMP without losing DEPTNOs 30 and 40 from DEPT.

The second solution moves the filtering into an inline view. Inline view E filters on EMP.DEPTNO and returns EMP rows of interest. These are then outer joined to DEPT. Because DEPT is the anchor table in the outer join, all departments, including 30 and 40, are returned.

11.4 Determining Which Rows Are Reciprocals

Problem

You have a table containing the results of two tests, and you want to determine which pair of scores are reciprocals. Consider the following result set from view V:

```
select *
  from V

TEST1      TEST2
----- ----------
   20         20
   50         25
   20         20
   60         30
   70         90
   80        130
   90         70
  100         50
  110         55
  120         60
  130         80
  140         70
```

Examining these results, you see that a test score for TEST1 of 70 and TEST2 of 90 is a reciprocal (there exists a score of 90 for TEST1 and a score of 70 for TEST2). Likewise, the scores of 80 for TEST1 and 130 for TEST2 are reciprocals of 130 for TEST1 and 80 for TEST2. Additionally, the scores of 20 for TEST1 and 20 for TEST2 are reciprocals of 20 for TEST2 and 20 for TEST1. You want to identify only one set of reciprocals. You want your result set to be this:

```
TEST1      TEST2
-----  ---------
   20         20
   70         90
   80        130
```

not this:

```
TEST1      TEST2
-----  ---------
   20         20
   20         20
   70         90
   80        130
   90         70
  130         80
```

Solution

Use a self-join to identify rows where TEST1 equals TEST2, and vice versa:

```
select distinct v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
   and v1.test2 = v2.test1
   and v1.test1 <= v1.test2
```

Discussion

The self-join results in a Cartesian product in which every TEST1 score can be compared against every TEST2 score, and vice versa. The following query will identify the reciprocals:

```
select v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
   and v1.test2 = v2.test1

TEST1      TEST2
----- ----------
   20         20
   20         20
   20         20
   20         20
```

```
      90         70
     130         80
      70         90
      80        130
```

The use of DISTINCT ensures that duplicate rows are removed from the final result set. The final filter in the WHERE clause (and V1.TEST1 <= V1.TEST2) will ensure that only one pair of reciprocals (where TEST1 is the smaller or equal value) is returned.

11.5 Selecting the Top n Records

Problem

You want to limit a result set to a specific number of records based on a ranking of some sort. For example, you want to return the names and salaries of the employees with the top five salaries.

Solution

The solution to this problem depends on the use of a window function. Which window function you will use depends on how you want to deal with ties. The following solution uses DENSE_RANK so that each tie in salary will count as only one against the total:

```
1  select ename,sal
2    from (
3  select ename, sal,
4         dense_rank() over (order by sal desc) dr
5    from emp
6         ) x
7   where dr <= 5
```

The total number of rows returned may exceed five, but there will be only five distinct salaries. Use ROW_NUMBER OVER if you want to return five rows regardless of ties (as no ties are allowed with this function).

Discussion

The window function DENSE_RANK OVER in inline view X does all the work. The following example shows the entire table after applying that function:

```
select ename, sal,
       dense_rank() over (order by sal desc) dr
  from emp

ENAME      SAL         DR
------- ------ ----------
KING      5000          1
```

```
SCOTT     3000          2
FORD      3000          2
JONES     2975          3
BLAKE     2850          4
CLARK     2450          5
ALLEN     1600          6
TURNER    1500          7
MILLER    1300          8
WARD      1250          9
MARTIN    1250          9
ADAMS     1100         10
JAMES      950         11
SMITH      800         12
```

Now it's just a matter of returning rows where DR is less than or equal to five.

11.6 Finding Records with the Highest and Lowest Values

Problem

You want to find "extreme" values in your table. For example, you want to find the employees with the highest and lowest salaries in table EMP.

Solution

DB2, Oracle, and SQL Server

Use the window functions MIN OVER and MAX OVER to find the lowest and highest salaries, respectively:

```
1  select ename
2    from (
3  select ename, sal,
4         min(sal)over() min_sal,
5         max(sal)over() max_sal
6    from emp
7         ) x
8   where sal in (min_sal,max_sal)
```

Discussion

DB2, Oracle, and SQL Server

The window functions MIN OVER and MAX OVER allow each row to have access to the lowest and highest salaries. The result set from inline view X is as follows:

```
select ename, sal,
       min(sal)over() min_sal,
       max(sal)over() max_sal
  from emp
```

```
ENAME      SAL    MIN_SAL    MAX_SAL
------- ------ ---------- ----------
SMITH      800        800       5000
ALLEN     1600        800       5000
WARD      1250        800       5000
JONES     2975        800       5000
MARTIN    1250        800       5000
BLAKE     2850        800       5000
CLARK     2450        800       5000
SCOTT     3000        800       5000
KING      5000        800       5000
TURNER    1500        800       5000
ADAMS     1100        800       5000
JAMES      950        800       5000
FORD      3000        800       5000
MILLER    1300        800       5000
```

Given this result set, all that's left is to return rows where SAL equals MIN_SAL or MAX_SAL.

11.7 Investigating Future Rows

Problem

You want to find any employees who earn less than the employee hired immediately after them. Based on the following result set:

```
ENAME             SAL HIREDATE
---------- ---------- ---------
SMITH             800 17-DEC-80
ALLEN            1600 20-FEB-81
WARD             1250 22-FEB-81
JONES            2975 02-APR-81
BLAKE            2850 01-MAY-81
CLARK            2450 09-JUN-81
TURNER           1500 08-SEP-81
MARTIN           1250 28-SEP-81
KING             5000 17-NOV-81
JAMES             950 03-DEC-81
FORD             3000 03-DEC-81
MILLER           1300 23-JAN-82
SCOTT            3000 09-DEC-82
ADAMS            1100 12-JAN-83
```

SMITH, WARD, MARTIN, JAMES, and MILLER earn less than the person hired immediately after they were hired, so those are the employees you want to find with a query.

Solution

The first step is to define what "future" means. You must impose order on your result set to be able to define a row as having a value that is "later" than another.

You can use the LEAD OVER window function to access the salary of the next employee that was hired. It's then a simple matter to check whether that salary is larger:

```
select ename, sal, hiredate
  from (
select ename, sal, hiredate,
       lead(sal)over(order by hiredate) next_sal
  from emp
       ) alias
 where sal < next_sal
```

Discussion

The window function LEAD OVER is perfect for a problem such as this one. It not only makes for a more readable query than the solution for the other products, LEAD OVER also leads to a more flexible solution because an argument can be passed to it that will determine how many rows ahead it should look (by default one). Being able to leap ahead more than one row is important in the case of duplicates in the column you are ordering by.

The following example shows how easy it is to use LEAD OVER to look at the salary of the "next" employee hired:

```
select ename, sal, hiredate,
       lead(sal)over(order by hiredate) next_sal
  from emp

ENAME      SAL HIREDATE    NEXT_SAL
------- ------ --------- ----------
SMITH      800 17-DEC-80       1600
ALLEN     1600 20-FEB-81       1250
WARD      1250 22-FEB-81       2975
JONES     2975 02-APR-81       2850
BLAKE     2850 01-MAY-81       2450
CLARK     2450 09-JUN-81       1500
TURNER    1500 08-SEP-81       1250
MARTIN    1250 28-SEP-81       5000
KING      5000 17-NOV-81        950
JAMES      950 03-DEC-81       3000
FORD      3000 03-DEC-81       1300
MILLER    1300 23-JAN-82       3000
SCOTT     3000 09-DEC-82       1100
ADAMS     1100 12-JAN-83
```

The final step is to return only rows where SAL is less than NEXT_SAL. Because of LEAD OVER's default range of one row, if there had been duplicates in table EMP—in particular, multiple employees hired on the same date—their SAL would be compared. This may or may not have been what you intended. If your goal is to compare the SAL of each employee with SAL of the next employee hired, excluding other employees hired on the same day, you can use the following solution as an alternative:

```
select ename, sal, hiredate
  from (
select ename, sal, hiredate,
       lead(sal,cnt-rn+1)over(order by hiredate) next_sal
  from (
select ename,sal,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
       )
       )
 where sal < next_sal
```

The idea behind this solution is to find the distance from the current row to the row it should be compared with. For example, if there are five duplicates, the first of the five needs to leap five rows to get to its correct LEAD OVER row. The value for CNT represents, for each employee with a duplicate HIREDATE, how many duplicates there are in total for their HIREDATE. The value for RN represents a ranking for the employees in DEPTNO 10. The rank is partitioned by HIREDATE so only employees with a HIREDATE that another employee has will have a value greater than one. The ranking is sorted by EMPNO (this is arbitrary). Now that you know how many total duplicates there are and you have a ranking of each duplicate, the distance to the next HIREDATE is simply the total number of duplicates minus the current rank plus one (CNT-RN+1).

See Also

For additional examples of using LEAD OVER in the presence of duplicates (and a more thorough discussion of this technique), see Recipe 8.7 and Recipe 10.2.

11.8 Shifting Row Values

Problem

You want to return each employee's name and salary along with the next highest and lowest salaries. If there are no higher or lower salaries, you want the results to wrap (first SAL shows last SAL and vice versa). You want to return the following result set:

```
ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800        950       5000
JAMES             950       1100        800
ADAMS            1100       1250        950
WARD             1250       1250       1100
MARTIN           1250       1300       1250
MILLER           1300       1500       1250
TURNER           1500       1600       1300
ALLEN            1600       2450       1500
CLARK            2450       2850       1600
BLAKE            2850       2975       2450
JONES            2975       3000       2850
SCOTT            3000       3000       2975
FORD             3000       5000       3000
KING             5000        800       3000
```

Solution

The window functions LEAD OVER and LAG OVER make this problem easy to solve and the resulting queries very readable. Use the window functions LAG OVER and LEAD OVER to access prior and next rows relative to the current row:

```
1  select ename,sal,
2         coalesce(lead(sal)over(order by sal),min(sal)over()) forward,
3         coalesce(lag(sal)over(order by sal),max(sal)over()) rewind
4    from emp
```

Discussion

The window functions LAG OVER and LEAD OVER will (by default and unless otherwise specified) return values from the row before and after the current row, respectively. You define what "before" or "after" means in the ORDER BY portion of the OVER clause. If you examine the solution, the first step is to return the next and prior rows relative to the current row, ordered by SAL:

```
select ename,sal,
       lead(sal)over(order by sal) forward,
       lag(sal)over(order by sal) rewind
  from emp


ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800        950
JAMES             950       1100        800
ADAMS            1100       1250        950
WARD             1250       1250       1100
MARTIN           1250       1300       1250
MILLER           1300       1500       1250
TURNER           1500       1600       1300
```

```
ALLEN            1600       2450       1500
CLARK            2450       2850       1600
BLAKE            2850       2975       2450
JONES            2975       3000       2850
SCOTT            3000       3000       2975
FORD             3000       5000       3000
KING             5000                  3000
```

Notice that REWIND is NULL for employee SMITH, and FORWARD is NULL for employee KING; that is because those two employees have the lowest and highest salaries, respectively. The requirement in the "Problem" section should NULL values exist in FORWARD or REWIND is to "wrap" the results, meaning that for the highest SAL, FORWARD should be the value of the lowest SAL in the table, and for the lowest SAL, REWIND should be the value of the highest SAL in the table. The window functions MIN OVER and MAX OVER with no partition or window specified (i.e., an empty parentheses after the OVER clause) will return the lowest and highest salaries in the table, respectively. The results are shown here:

```
select ename,sal,
       coalesce(lead(sal)over(order by sal),min(sal)over()) forward,
       coalesce(lag(sal)over(order by sal),max(sal)over()) rewind
  from emp


ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800        950       5000
JAMES             950       1100        800
ADAMS            1100       1250        950
WARD             1250       1250       1100
MARTIN           1250       1300       1250
MILLER           1300       1500       1250
TURNER           1500       1600       1300
ALLEN            1600       2450       1500
CLARK            2450       2850       1600
BLAKE            2850       2975       2450
JONES            2975       3000       2850
SCOTT            3000       3000       2975
FORD             3000       5000       3000
KING             5000        800       3000
```

Another useful feature of LAG OVER and LEAD OVER is the ability to define how far forward or back you would like to go. In the example for this recipe, you go only one row forward or back. If want to move three rows forward and five rows back, doing so is simple. Just specify the values 3 and 5, as shown here:

```
select ename,sal,
       lead(sal,3)over(order by sal) forward,
       lag(sal,5)over(order by sal) rewind
  from emp
```

```
ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800       1250
JAMES             950       1250
ADAMS            1100       1300
WARD             1250       1500
MARTIN           1250       1600
MILLER           1300       2450        800
TURNER           1500       2850        950
ALLEN            1600       2975       1100
CLARK            2450       3000       1250
BLAKE            2850       3000       1250
JONES            2975       5000       1300
SCOTT            3000                  1500
FORD             3000                  1600
KING             5000                  2450
```

11.9 Ranking Results

Problem

You want to rank the salaries in table EMP while allowing for ties. You want to return the following result set:

```
RNK     SAL
--- -------
  1     800
  2     950
  3    1100
  4    1250
  4    1250
  5    1300
  6    1500
  7    1600
  8    2450
  9    2850
 10    2975
 11    3000
 11    3000
 12    5000
```

Solution

Window functions make ranking queries extremely simple. Three window functions are particularly useful for ranking: DENSE_RANK OVER, ROW_NUMBER OVER, and RANK OVER.

Because you want to allow for ties, use the window function DENSE_RANK OVER:

```
1 select dense_rank() over(order by sal) rnk, sal
2   from emp
```

Discussion

The window function DENSE_RANK OVER does all the legwork here. In parentheses following the OVER keyword you place an ORDER BY clause to specify the order in which rows are ranked. The solution uses ORDER BY SAL, so rows from EMP are ranked in ascending order of salary.

11.10 Suppressing Duplicates

Problem

You want to find the different job types in table EMP but do not want to see duplicates. The result set should be as follows:

```
JOB
---------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

Solution

All of the RDBMSs support the keyword DISTINCT, and it arguably is the easiest mechanism for suppressing duplicates from the result set. However, this recipe will also cover two additional methods for suppressing duplicates.

The traditional method of using DISTINCT and sometimes GROUP BY certainly works. The following solution is an alternative that makes use of the window function ROW_NUMBER OVER:

```
select job
  from (
select job,
       row_number()over(partition by job order by job) rn
  from emp
       ) x
 where rn = 1
```

Traditional alternatives

Use the DISTINCT keyword to suppress duplicates from the result set:

```
select distinct job
  from emp
```

Additionally, it is also possible to use GROUP BY to suppress duplicates:

```
select job
  from emp
 group by job
```

Discussion

DB2, Oracle, and SQL Server

This solution depends on some outside-the-box thinking about partitioned window functions. By using PARTITION BY in the OVER clause of ROW_NUMBER, you can reset the value returned by ROW_NUMBER to 1 whenever a new job is encountered. The following results are from inline view X:

```
select job,
       row_number()over(partition by job order by job) rn
  from emp


JOB               RN
--------- ----------
ANALYST            1
ANALYST            2
CLERK              1
CLERK              2
CLERK              3
CLERK              4
MANAGER            1
MANAGER            2
MANAGER            3
PRESIDENT          1
SALESMAN           1
SALESMAN           2
SALESMAN           3
SALESMAN           4
```

Each row is given an increasing, sequential number, and that number is reset to one whenever the job changes. To filter out the duplicates, all you must do is keep the rows where RN is 1.

An ORDER BY clause is mandatory when using ROW_NUMBER OVER (except in DB2) but doesn't affect the result. Which job is returned is irrelevant so long as you return one of each job.

Traditional alternatives

The first solution shows how to use the keyword DISTINCT to suppress duplicates from a result set. Keep in mind that DISTINCT is applied to the whole SELECT list;

additional columns can and will change the result set. Consider the difference between these two queries:

```
select distinct job
  from emp

JOB
---------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

```
select distinct job, deptno
  from emp

JOB           DEPTNO
--------- ----------
ANALYST           20
CLERK             10
CLERK             20
CLERK             30
MANAGER           10
MANAGER           20
MANAGER           30
PRESIDENT         10
SALESMAN          30
```

By adding DEPTNO to the SELECT list, what you return is each DISTINCT pair of JOB/DEPTNO values from table EMP.

The second solution uses GROUP BY to suppress duplicates. While using GROUP BY in this way is not uncommon, keep in mind that GROUP BY and DISTINCT are two very different clauses that are not interchangeable. I've included GROUP BY in this solution for completeness, as you will no doubt come across it at some point.

11.11 Finding Knight Values

Problem

You want return a result set that contains each employee's name, the department they work in, their salary, the date they were hired, and the salary of the last employee hired, in each department. You want to return the following result set:

```
DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER            1300 23-JAN-2007      1300
    10 KING              5000 17-NOV-2006      1300
    10 CLARK             2450 09-JUN-2006      1300
    20 ADAMS             1100 12-JAN-2007      1100
    20 SCOTT             3000 09-DEC-2007      1100
    20 FORD              3000 03-DEC-2006      1100
    20 JONES             2975 02-APR-2006      1100
    20 SMITH              800 17-DEC-2005      1100
    30 JAMES              950 03-DEC-2006       950
    30 MARTIN            1250 28-SEP-2006       950
    30 TURNER            1500 08-SEP-2006       950
    30 BLAKE             2850 01-MAY-2006       950
    30 WARD              1250 22-FEB-2006       950
    30 ALLEN             1600 20-FEB-2006       950
```

The values in LATEST_SAL are the "knight values" because the path to find them is analogous to a knight's path in the game of chess. You determine the result the way a knight determines a new location: by jumping to a row and then turning and jumping to a different column (see Figure 11-1). To find the correct values for LATEST_SAL, you must first locate (jump to) the row with the latest HIREDATE in each DEPTNO, and then you select (jump to) the SAL column of that row.

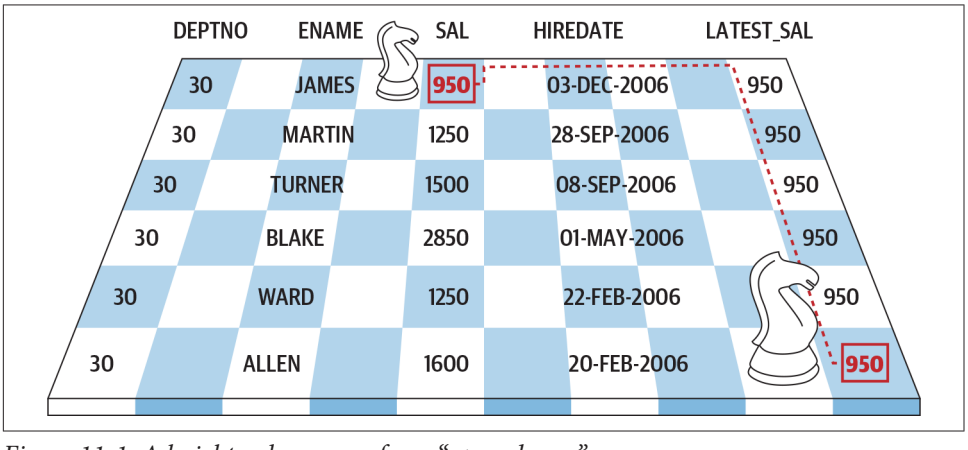


Figure 11-1. A knight value comes from "up and over"

The term *knight value* was coined by a clever coworker of Anthony's, Kay Young. After having him review the recipes for correctness, Anthony admitted to Kay that he was stumped and could not come up with a good title. Because you need to initially evaluate one row and then "jump" and take a value from another, Kay came up with the term *knight value*.

Solution

DB2 and SQL Server

Use a CASE expression in a subquery to return the SAL of the last employee hired in each DEPTNO; for all other salaries, return 0. Use the window function MAX OVER in the outer query to return the nonzero SAL for each employee's department:

```
select deptno,
       ename,
       sal,
       hiredate,
       max(latest_sal)over(partition by deptno) latest_sal
  from (
select deptno,
       ename,
       sal,
```

```
10         hiredate,
11         case
12           when hiredate = max(hiredate)over(partition by deptno)
13           then sal else 0
14         end latest_sal
15    from emp
16         ) x
17   order by 1, 4 desc
```

Oracle

Use the window function MAX OVER to return the highest SAL for each DEPTNO. Use the functions DENSE_RANK and LAST, while ordering by HIREDATE in the KEEP clause to return the highest SAL for the latest HIREDATE in a given DEPTNO:

```
1  select deptno,
2          ename,
3          sal,
4          hiredate,
5           max(sal)
6             keep(dense_rank last order by hiredate)
7             over(partition by deptno) latest_sal
8    from emp
9  order by 1, 4 desc
```

Discussion

DB2 and SQL Server

The first step is to use the window function MAX OVER in a CASE expression to find the employee hired last, or most recently, in each DEPTNO. If an employee's HIREDATE matches the value returned by MAX OVER, then use a CASE expression to return that employee's SAL; otherwise, return zero. The results of this are shown here:

```
select deptno,
       ename,
       sal,
       hiredate,
       case
           when hiredate = max(hiredate)over(partition by deptno)
           then sal else 0
       end latest_sal
  from emp


DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ --------- ----------- ----------- ----------
    10 CLARK            2450 09-JUN-2006          0
    10 KING             5000 17-NOV-2006          0
    10 MILLER           1300 23-JAN-2007       1300
```

```
    20 SMITH             800 17-DEC-2005          0
    20 ADAMS            1100 12-JAN-2007       1100
    20 FORD             3000 03-DEC-2006          0
    20 SCOTT            3000 09-DEC-2007          0
    20 JONES            2975 02-APR-2006          0
    30 ALLEN            1600 20-FEB-2006          0
    30 BLAKE            2850 01-MAY-2006          0
    30 MARTIN           1250 28-SEP-2006          0
    30 JAMES             950 03-DEC-2006        950
    30 TURNER           1500 08-SEP-2006          0
    30 WARD             1250 22-FEB-2006          0
```

Because the value for LATEST_SAL will be either zero or the SAL of the employee(s) hired most recently, you can wrap the previous query in an inline view and use MAX OVER again, but this time to return the greatest nonzero LATEST_SAL for each DEPTNO:

```
select deptno,
       ename,
       sal,
       hiredate,
       max(latest_sal)over(partition by deptno) latest_sal
  from (
select deptno,
       ename,
       sal,
       hiredate,
       case
           when hiredate = max(hiredate)over(partition by deptno)
           then sal else 0
       end latest_sal
  from emp
       ) x
 order by 1, 4 desc


DEPTNO  ENAME            SAL HIREDATE    LATEST_SAL
------- --------- ---------- ----------- ----------
    10  MILLER          1300 23-JAN-2007       1300
    10  KING            5000 17-NOV-2006       1300
    10  CLARK           2450 09-JUN-2006       1300
    20  ADAMS           1100 12-JAN-2007       1100
    20  SCOTT           3000 09-DEC-2007       1100
    20  FORD            3000 03-DEC-2006       1100
    20  JONES           2975 02-APR-2006       1100
    20  SMITH            800 17-DEC-2005       1100
    30  JAMES            950 03-DEC-2006        950
    30  MARTIN          1250 28-SEP-2006        950
    30  TURNER          1500 08-SEP-2006        950
    30  BLAKE           2850 01-MAY-2006        950
    30  WARD            1250 22-FEB-2006        950
    30  ALLEN           1600 20-FEB-2006        950
```

Oracle

The key to the Oracle solution is to take advantage of the KEEP clause. The KEEP clause allows you to rank the rows returned by a group/partition and work with the first or last row in the group. Consider what the solution looks like without KEEP:

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal) over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc


DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-2007       5000
    10 KING             5000 17-NOV-2006       5000
    10 CLARK            2450 09-JUN-2006       5000
    20 ADAMS            1100 12-JAN-2007       3000
    20 SCOTT            3000 09-DEC-2007       3000
    20 FORD             3000 03-DEC-2006       3000
    20 JONES            2975 02-APR-2006       3000
    20 SMITH             800 17-DEC-2005       3000
    30 JAMES             950 03-DEC-2006       2850
    30 MARTIN           1250 28-SEP-2006       2850
    30 TURNER           1500 08-SEP-2006       2850
    30 BLAKE            2850 01-MAY-2006       2850
    30 WARD             1250 22-FEB-2006       2850
    30 ALLEN            1600 20-FEB-2006       2850
```

Rather than returning the SAL of the latest employee hired, MAX OVER without KEEP simply returns the highest salary in each DEPTNO. KEEP, in this recipe, allows you to order the salaries by HIREDATE in each DEPTNO by specifying ORDER BY HIREDATE. Then, the function DENSE_RANK assigns a rank to each HIREDATE in ascending order. Finally, the function LAST determines which row to apply the aggregate function to: the "last" row based on the ranking of DENSE_RANK. In this case, the aggregate function MAX is applied to the SAL column for the row with the "last" HIREDATE. In essence, keep the SAL of the HIREDATE ranked last in each DEPTNO.

You are ranking the rows in each DEPTNO based on one column (HIREDATE), but then applying the aggregation (MAX) on another column (SAL). This ability to rank in one dimension and aggregate over another is convenient as it allows you to avoid extra joins and inline views as are used in the other solutions. Finally, by adding the OVER clause after the KEEP clause, you can return the SAL "kept" by KEEP for each row in the partition.

Alternatively, you can order by HIREDATE in descending order and "keep" the first SAL. Compare the following two queries, which return the same result set:

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank last order by hiredate)
         over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc


DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-2007       1300
    10 KING             5000 17-NOV-2006       1300
    10 CLARK            2450 09-JUN-2006       1300
    20 ADAMS            1100 12-JAN-2007       1100
    20 SCOTT            3000 09-DEC-2007       1100
    20 FORD             3000 03-DEC-2006       1100
    20 JONES            2975 02-APR-2006       1100
    20 SMITH             800 17-DEC-2005       1100
    30 JAMES             950 03-DEC-2006        950
    30 MARTIN           1250 28-SEP-2006        950
    30 TURNER           1500 08-SEP-2006        950
    30 BLAKE            2850 01-MAY-2006        950
    30 WARD             1250 22-FEB-2006        950
    30 ALLEN            1600 20-FEB-2006        950


select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank first order by hiredate desc)
         over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc


DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-2007       1300
    10 KING             5000 17-NOV-2006       1300
    10 CLARK            2450 09-JUN-2006       1300
    20 ADAMS            1100 12-JAN-2007       1100
    20 SCOTT            3000 09-DEC-2007       1100
    20 FORD             3000 03-DEC-2006       1100
    20 JONES            2975 02-APR-2006       1100
```

```
20 SMITH             800 17-DEC-2005       1100
30 JAMES             950 03-DEC-2006        950
30 MARTIN           1250 28-SEP-2006        950
30 TURNER           1500 08-SEP-2006        950
30 BLAKE            2850 01-MAY-2006        950
30 WARD             1250 22-FEB-2006        950
30 ALLEN            1600 20-FEB-2006        950
```

11.12 Generating Simple Forecasts

Problem

Based on current data, you want to return additional rows and columns representing future actions. For example, consider the following result set:

```
ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005  27-SEP-2005
 2 26-SEP-2005  28-SEP-2005
 3 27-SEP-2005  29-SEP-2005
```

You want to return three rows per row returned in your result set (each row plus two additional rows for each order). Along with the extra rows, you would like to return two additional columns providing dates for expected order processing.

From the previous result set, you can see that an order takes two days to process. For the purposes of this example, let's say the next step after processing is verification, and the last step is shipment. Verification occurs one day after processing, and shipment occurs one day after verification. You want to return a result set expressing the whole procedure. Ultimately you want to transform the previous result set to the following result set:

```
ID ORDER_DATE  PROCESS_DATE  VERIFIED     SHIPPED
-- ----------- ------------  -----------  -----------
 1 25-SEP-2005  27-SEP-2005
 1 25-SEP-2005  27-SEP-2005  28-SEP-2005
 1 25-SEP-2005  27-SEP-2005  28-SEP-2005  29-SEP-2005
 2 26-SEP-2005  28-SEP-2005
 2 26-SEP-2005  28-SEP-2005  29-SEP-2005
 2 26-SEP-2005  28-SEP-2005  29-SEP-2005  30-SEP-2005
 3 27-SEP-2005  29-SEP-2005
 3 27-SEP-2005  29-SEP-2005  30-SEP-2005
 3 27-SEP-2005  29-SEP-2005  30-SEP-2005  01-OCT-2005
```

Solution

The key is to use a Cartesian product to generate two additional rows for each order and then simply use CASE expressions to create the required column values.

DB2, MySQL, and SQL Server

Use the recursive WITH clause to generate rows needed for your Cartesian product. The DB2 and SQL Server solutions are identical except for the function used to retrieve the current date. DB2 uses CURRENT_DATE and SQL Server uses GETDATE. MySQL uses the CURDATE and requires the insertion of the keyword RECURSIVE after WITH to indicate that this is a recursive CTE. The SQL Server solution is shown here:

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
            then process_date+1
            else null
       end as verified,
       case when nrows.n = 3
            then process_date+2
            else null
       end as shipped
  from (
select nrows.n id,
       getdate()+nrows.n   as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows
       ) orders, nrows
 order by 1
```

Oracle

Use the hierarchical CONNECT BY clause to generate the three rows needed for the Cartesian product. Use the WITH clause to allow you to reuse the results returned by CONNECT BY without having to call it again:

```
with nrows as (
select level n
  from dual
connect by level <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
            then process_date+1
            else null
       end as verified,
       case when nrows.n = 3
```

```
14             then process_date+2
15             else null
16        end as shipped
17  from (
18 select nrows.n id,
19       sysdate+nrows.n as order_date,
20       sysdate+nrows.n+2 as process_date
21   from nrows
22       ) orders, nrows
```

PostgreSQL

You can create a Cartesian product many different ways; this solution uses the PostgreSQL function GENERATE_SERIES:

```
 1 select id,
 2        order_date,
 3        process_date,
 4        case when gs.n >= 2
 5             then process_date+1
 6             else null
 7        end as verified,
 8        case when gs.n = 3
 9             then process_date+2
10             else null
11        end as shipped
12  from (
13 select gs.id,
14        current_date+gs.id as order_date,
15        current_date+gs.id+2 as process_date
16   from generate_series(1,3) gs (id)
17        ) orders,
18          generate_series(1,3)gs(n)
```

MySQL

MySQL does not support a function for automatic row generation.

Discussion

DB2, MySQL, and SQL Server

The result set presented in the "Problem" section is returned via inline view ORDERS, and is shown here:

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select nrows.n id,getdate()+nrows.n   as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows
```

```
ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005  27-SEP-2005
 2 26-SEP-2005  28-SEP-2005
 3 27-SEP-2005  29-SEP-2005
```

This query simply uses the WITH clause to make up three rows representing the orders you must process. NROWS returns the values 1, 2, and 3, and those numbers are added to GETDATE (CURRENT_DATE for DB2, CURDATE() for MySQL) to represent the dates of the orders. Because the "Problem" section states that processing time takes two days, the query also adds two days to the ORDER_DATE (adds the value returned by NROWS to GETDATE and then adds two more days).

Now that you have your base result set, the next step is to create a Cartesian product because the requirement is to return three rows for each order. Use NROWS to create a Cartesian product to return three rows for each order:

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select nrows.n,
       orders.*
  from (
select nrows.n id,
       getdate()+nrows.n    as order_date,
        getdate()+nrows.n+2 as process_date
  from nrows
       ) orders, nrows
 order by 2,1

  N  ID  ORDER_DATE  PROCESS_DATE
--- ---  ----------- ------------
  1   1  25-SEP-2005  27-SEP-2005
  2   1  25-SEP-2005  27-SEP-2005
  3   1  25-SEP-2005  27-SEP-2005
  1   2  26-SEP-2005  28-SEP-2005
  2   2  26-SEP-2005  28-SEP-2005
  3   2  26-SEP-2005  28-SEP-2005
  1   3  27-SEP-2005  29-SEP-2005
  2   3  27-SEP-2005  29-SEP-2005
  3   3  27-SEP-2005  29-SEP-2005
```

Now that you have three rows for each order, simply use a CASE expression to create the addition column values to represent the status of verification and shipment.

The first row for each order should have a NULL value for VERIFIED and SHIPPED. The second row for each order should have a NULL value for SHIPPED. The third row for each order should have non-NULL values for each column. The final result set is shown here:

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
            then process_date+1
            else null

       end as verified,
       case when nrows.n = 3
           then process_date+2
           else null
       end as shipped
  from (
select nrows.n id,
       getdate()+nrows.n   as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows
       ) orders, nrows
 order by 1

ID ORDER_DATE  PROCESS_DATE  VERIFIED     SHIPPED
-- ----------- ------------  -----------  -----------
 1 25-SEP-2005  27-SEP-2005
 1 25-SEP-2005  27-SEP-2005  28-SEP-2005
 1 25-SEP-2005  27-SEP-2005  28-SEP-2005  29-SEP-2005
 2 26-SEP-2005  28-SEP-2005
 2 26-SEP-2005  28-SEP-2005  29-SEP-2005
 2 26-SEP-2005  28-SEP-2005  29-SEP-2005  30-SEP-2005
 3 27-SEP-2005  29-SEP-2005
 3 27-SEP-2005  29-SEP-2005  30-SEP-2005
 3 27-SEP-2005  29-SEP-2005  30-SEP-2005  01-OCT-2005
```

The final result set expresses the complete order process, from the day the order was received to the day it should be shipped.

Oracle

The result set presented in the problem section is returned via inline view ORDERS and is shown here:

```
with nrows as (
select level n
  from dual
```

```
connect by level <= 3
)
select nrows.n id,
       sysdate+nrows.n order_date,
       sysdate+nrows.n+2 process_date
  from nrows

ID ORDER_DATE   PROCESS_DATE
-- -----------  ------------
 1 25-SEP-2005   27-SEP-2005
 2 26-SEP-2005   28-SEP-2005
 3 27-SEP-2005   29-SEP-2005
```

This query simply uses CONNECT BY to make up three rows representing the orders you must process. Use the WITH clause to refer to the rows returned by CONNECT BY as NROWS.N. CONNECT BY returns the values 1, 2, and 3, and those numbers are added to SYSDATE to represent the dates of the orders. Since the "Problem" section states that processing time takes two days, the query also adds two days to the ORDER_DATE (adds the value returned by GENERATE_ SERIES to SYSDATE and then adds two more days).

Now that you have your base result set, the next step is to create a Cartesian product because the requirement is to return three rows for each order. Use NROWS to create a Cartesian product to return three rows for each order:

```
with nrows as (
select level n
  from dual
connect by level <= 3
)
select nrows.n,
       orders.*
  from (
select nrows.n id,
       sysdate+nrows.n order_date,
       sysdate+nrows.n+2 process_date
  from nrows
  ) orders, nrows

  N  ID ORDER_DATE  PROCESS_DATE
--- --- ----------- ------------
  1   1 25-SEP-2005  27-SEP-2005
  2   1 25-SEP-2005  27-SEP-2005
  3   1 25-SEP-2005  27-SEP-2005
  1   2 26-SEP-2005  28-SEP-2005
  2   2 26-SEP-2005  28-SEP-2005
  3   2 26-SEP-2005  28-SEP-2005
  1   3 27-SEP-2005  29-SEP-2005
  2   3 27-SEP-2005  29-SEP-2005
  3   3 27-SEP-2005  29-SEP-2005
```

Now that you have three rows for each order, simply use a CASE expression to create the addition column values to represent the status of verification and shipment.

The first row for each order should have a NULL value for VERIFIED and SHIPPED. The second row for each order should have a NULL value for SHIPPED. The third row for each order should have non-NULL values for each column. The final result set is shown here:

```
with nrows as (
select level n
  from dual
connect by level <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
            then process_date+1
            else null
       end as verified,
       case when nrows.n = 3
            then process_date+2
            else null
       end as shipped
  from (
select nrows.n id,
       sysdate+nrows.n order_date,
       sysdate+nrows.n+2 process_date
  from nrows
       ) orders, nrows

 ID ORDER_DATE  PROCESS_DATE  VERIFIED     SHIPPED
 -- ----------- ------------  -----------  -----------
  1 25-SEP-2005  27-SEP-2005
  1 25-SEP-2005  27-SEP-2005  28-SEP-2005
  1 25-SEP-2005  27-SEP-2005  28-SEP-2005  29-SEP-2005
  2 26-SEP-2005  28-SEP-2005
  2 26-SEP-2005  28-SEP-2005  29-SEP-2005
  2 26-SEP-2005  28-SEP-2005  29-SEP-2005  30-SEP-2005
  3 27-SEP-2005  29-SEP-2005
  3 27-SEP-2005  29-SEP-2005  30-SEP-2005
  3 27-SEP-2005  29-SEP-2005  30-SEP-2005  01-OCT-2005
```

The final result set expresses the complete order process from the day the order was received to the day it should be shipped.

PostgreSQL

The result set presented in the problem section is returned via inline view ORDERS and is shown here:

```
select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
 from generate_series(1,3) gs (id)

ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005  27-SEP-2005
 2 26-SEP-2005  28-SEP-2005
 3 27-SEP-2005  29-SEP-2005
```

This query simply uses the GENERATE_SERIES function to make up three rows representing the orders you must process. GENERATE_SERIES returns the values 1, 2, and 3, and those numbers are added to CURRENT_DATE to represent the dates of the orders. Since the "Problem" section states that processing time takes two days, the query also adds two days to the ORDER_DATE (adds the value returned by GENERATE_SERIES to CURRENT_DATE and then adds two more days). Now that you have your base result set, the next step is to create a Cartesian product because the requirement is to return three rows for each order. Use the GENERATE_ SERIES function to create a Cartesian product to return three rows for each order:

```
select gs.n,
       orders.*
  from (
select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs (id)
       ) orders,
         generate_series(1,3)gs(n)

  N ID  ORDER_DATE  PROCESS_DATE
--- --- ----------- ------------
  1   1 25-SEP-2005  27-SEP-2005
  2   1 25-SEP-2005  27-SEP-2005
  3   1 25-SEP-2005  27-SEP-2005
  1   2 26-SEP-2005  28-SEP-2005
  2   2 26-SEP-2005  28-SEP-2005
  3   2 26-SEP-2005  28-SEP-2005
  1   3 27-SEP-2005  29-SEP-2005
  2   3 27-SEP-2005  29-SEP-2005
  3   3 27-SEP-2005  29-SEP-2005
```

Now that you have three rows for each order, simply use a CASE expression to create the addition column values to represent the status of verification and shipment.

The first row for each order should have a NULL value for VERIFIED and SHIPPED. The second row for each order should have a NULL value for SHIPPED. The third row for each order should have non-NULL values for each column. The final result set is shown here:

```
select id,
       order_date,
       process_date,
       case when gs.n >= 2
            then process_date+1
            else null
       end as verified,
       case when gs.n = 3
            then process_date+2
            else null
       end as shipped
  from (
select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs(id)
       ) orders,
         generate_series(1,3)gs(n)

ID ORDER_DATE  PROCESS_DATE  VERIFIED     SHIPPED
-- ----------- ------------  -----------  -----------
 1 25-SEP-2005 27-SEP-2005
 1 25-SEP-2005 27-SEP-2005   28-SEP-2005
 1 25-SEP-2005 27-SEP-2005   28-SEP-2005  29-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 2 26-SEP-2005 28-SEP-2005   29-SEP-2005
 2 26-SEP-2005 28-SEP-2005   29-SEP-2005  30-SEP-2005
 3 27-SEP-2005 29-SEP-2005
 3 27-SEP-2005 29-SEP-2005   30-SEP-2005
 3 27-SEP-2005 29-SEP-2005   30-SEP-2005  01-OCT-2005
```

The final result set expresses the complete order process from the day the order was received to the day it should be shipped.

11.13 Summing Up

The recipes from this chapter represent practical problems that can't be solved with a single function. They are some the kinds of problems that business users will frequently look to you to solve for them.

CHAPTER 12

Reporting and Reshaping

This chapter introduces queries you may find helpful for creating reports. These typically involve reporting-specific formatting considerations along with different levels of aggregation. Another focus of this chapter is transposing or pivoting result sets: reshaping the data by turning rows into columns.

In general, these recipes have in common that they allow you to present data in formats or shapes different from the way they are stored. As your comfort level with pivoting increases, you'll undoubtedly find uses for it outside of what are presented in this chapter.

12.1 Pivoting a Result Set into One Row

Problem

You want to take values from groups of rows and turn those values into columns in a single row per group. For example, you have a result set displaying the number of employees in each department:

```
DEPTNO        CNT
------ ----------
    10          3
    20          5
    30          6
```

You would like to reformat the output so that the result set looks as follows:

```
DEPTNO_10   DEPTNO_20   DEPTNO_30
---------  ----------  ----------
        3           5           6
```

This is a classic example of data presented in a different shape than the way it is stored.

Solution

Transpose the result set using CASE and the aggregate function SUM:

```
1 select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
2        sum(case when deptno=20 then 1 else 0 end) as deptno_20,
3        sum(case when deptno=30 then 1 else 0 end) as deptno_30
4   from emp
```

Discussion

This example is an excellent introduction to pivoting. The concept is simple: for each row returned by the unpivoted query, use a CASE expression to separate the rows into columns. Then, because this particular problem is to count the number of employees per department, use the aggregate function SUM to count the occurrence of each DEPTNO. If you're having trouble understanding how this works exactly, execute the query with the aggregate function SUM and include DEPTNO for readability:

```
select deptno,
       case when deptno=10 then 1 else 0 end as deptno_10,
       case when deptno=20 then 1 else 0 end as deptno_20,
       case when deptno=30 then 1 else 0 end as deptno_30
  from emp
 order by 1

 DEPTNO   DEPTNO_10   DEPTNO_20   DEPTNO_30
 ------  ----------  ----------  ----------
     10           1           0           0
     10           1           0           0
     10           1           0           0
     20           0           1           0
     20           0           1           0
     20           0           1           0
     20           0           1           0
     30           0           0           1
     30           0           0           1
     30           0           0           1
     30           0           0           1
     30           0           0           1
     30           0           0           1
```

You can think of each CASE expression as a flag to determine which DEPTNO a row belongs to. At this point, the "rows to columns" transformation is already done; the next step is to simply sum the values returned by DEPTNO_10, DEPTNO_20, and DEPTNO_30, and then to group by DEPTNO. The following are the results:

```
select deptno,
       sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp
 group by deptno

DEPTNO   DEPTNO_10   DEPTNO_20   DEPTNO_30
------  ----------  ----------  ----------
    10           3           0           0
    20           0           5           0
    30           0           0           6
```

If you inspect this result set, you see that logically the output makes sense; for example, DEPTNO 10 has three employees in DEPTNO_10 and zero in the other departments. Since the goal is to return one row, the last step is to remove the DEPTNO and GROUP BY clause and simply sum the CASE expressions:

```
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp

  DEPTNO_10   DEPTNO_20   DEPTNO_30
  ---------  ----------  ----------
          3           5           6
```

The following is another approach that you may sometimes see applied to this same sort of problem:

```
select max(case when deptno=10 then empcount else null end) as deptno_10
       max(case when deptno=20 then empcount else null end) as deptno_20,
       max(case when deptno=10 then empcount else null end) as deptno_30
  from (
select deptno, count(*) as empcount
  from emp
 group by deptno
       ) x
```

This approach uses an inline view to generate the employee counts per department. CASE expressions in the main query translate rows to columns, getting you to the following results:

```
DEPTNO_10   DEPTNO_20   DEPTNO_30
---------  ----------  ----------
        3        NULL        NULL
     NULL           5        NULL
     NULL        NULL           6
```

Then the MAX functions collapses the columns into one row:

```
DEPTNO_10   DEPTNO_20   DEPTNO_30
---------  ----------  ----------
        3           5           6
```

12.2 Pivoting a Result Set into Multiple Rows

Problem

You want to turn rows into columns by creating a column corresponding to each of the values in a single given column. However, unlike in the previous recipe, you need multiple rows of output. Like the earlier recipe, pivoting into multiple rows is a fundamental method of reshaping data.

For example, you want to return each employee and their position (JOB), and you currently use a query that returns the following result set:

```
JOB        ENAME
---------  ----------
ANALYST    SCOTT
ANALYST    FORD
CLERK      SMITH
CLERK      ADAMS
CLERK      MILLER
CLERK      JAMES
MANAGER    JONES
MANAGER    CLARK
MANAGER    BLAKE
PRESIDENT  KING
SALESMAN   ALLEN
SALESMAN   MARTIN
SALESMAN   TURNER
SALESMAN   WARD
```

You would like to format the result set such that each job gets its own column:

```
CLERKS  ANALYSTS  MGRS   PREZ  SALES
------  --------  -----  ----  ------
MILLER  FORD      CLARK  KING  TURNER
JAMES   SCOTT     BLAKE        MARTIN
ADAMS             JONES        WARD
SMITH                          ALLEN
```

Solution

Unlike the first recipe in this chapter, the result set for this recipe consists of more than one row. Using the previous recipe's technique will not work for this recipe, as the MAX(ENAME) for each JOB would be returned, which would result in one ENAME for each JOB (i.e., one row will be returned as in the first recipe). To solve

this problem, you must make each JOB/ENAME combination unique. Then, when you apply an aggregate function to remove NULLs, you don't lose any ENAMEs.

Use the ranking function ROW_NUMBER OVER to make each JOB/ENAME combination unique. Pivot the result set using a CASE expression and the aggregate function MAX while grouping on the value returned by the window function:

```
 1  select max(case when job='CLERK'
 2                  then ename else null end) as clerks,
 3         max(case when job='ANALYST'
 4                  then ename else null end) as analysts,
 5         max(case when job='MANAGER'
 6                  then ename else null end) as mgrs,
 7         max(case when job='PRESIDENT'
 8                  then ename else null end) as prez,
 9         max(case when job='SALESMAN'
10                  then ename else null end) as sales
11   from (
12 select job,
13        ename,
14        row_number()over(partition by job order by ename) rn
15   from emp
16        ) x
17  group by rn
```

Discussion

The first step is to use the window function ROW_NUMBER OVER to help make each JOB/ENAME combination unique:

```
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp

  JOB       ENAME              RN
  --------- ---------- ----------
  ANALYST   FORD                1
  ANALYST   SCOTT               2
  CLERK     ADAMS               1
  CLERK     JAMES               2
  CLERK     MILLER              3
  CLERK     SMITH               4
  MANAGER   BLAKE               1
  MANAGER   CLARK               2
  MANAGER   JONES               3
  PRESIDENT KING                1
  SALESMAN  ALLEN               1
  SALESMAN  MARTIN              2
  SALESMAN  TURNER              3
  SALESMAN  WARD                4
```

Giving each ENAME a unique "row number" within a given job prevents any problems that might otherwise result from two employees having the same name and job. The goal here is to be able to group on row number (on RN) without dropping any employees from the result set due to the use of MAX. This step is the most important step in solving the problem. Without this first step, the aggregation in the outer query will remove necessary rows. Consider what the result set would look like without using ROW_NUMBER OVER, using the same technique as shown in the first recipe:

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from emp


CLERKS      ANALYSTS    MGRS        PREZ        SALES
----------  ----------  ----------  ----------  ----------
SMITH       SCOTT       JONES       KING        WARD
```

Unfortunately, only one row is returned for each JOB: the employee with the MAX ENAME. When it comes time to pivot the result set, using MIN or MAX should serve as a means to remove NULLs from the result set, not restrict the ENAMEs returned. How this works will be come clearer as you continue through the explanation.

The next step uses a CASE expression to organize the ENAMEs into their proper column (JOB):

```
select rn,
       case when job='CLERK'
            then ename else null end as clerks,
       case when job='ANALYST'
            then ename else null end as analysts,
       case when job='MANAGER'
            then ename else null end as mgrs,
       case when job='PRESIDENT'
            then ename else null end as prez,
       case when job='SALESMAN'
            then ename else null end as sales
  from (
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp
       ) x
```

```
RN  CLERKS      ANALYSTS    MGRS        PREZ        SALES
--  ----------  ----------  ----------  ----------  ----------
 1              FORD
 2              SCOTT
 1  ADAMS
 2  JAMES
 3  MILLER
 4  SMITH
 1                          BLAKE
 2                          CLARK
 3                          JONES
 1                                      KING
 1                                                  ALLEN
 2                                                  MARTIN
 3                                                  TURNER
 4                                                  WARD
```

At this point, the rows are transposed into columns, and the last step is to remove the NULLs to make the result set more readable. To remove the NULLs, use the aggregate function MAX and group by RN. (You can use the function MIN as well. The choice to use MAX is arbitrary, as you will only ever be aggregating one value per group.) There is only one value for each RN/JOB/ENAME combination. Grouping by RN in conjunction with the CASE expressions embedded within the calls to MAX ensures that each call to MAX results in picking only one name from a group of otherwise NULL values:

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from (
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp
       ) x
group by rn

CLERKS  ANALYSTS  MGRS   PREZ  SALES
------  --------  -----  ----  ------
MILLER  FORD      CLARK  KING  TURNER
JAMES   SCOTT     BLAKE        MARTIN
ADAMS             JONES        WARD
SMITH                          ALLEN
```

The technique of using ROW_NUMBER OVER to create unique combinations of rows is extremely useful for formatting query results. Consider the following query that creates a sparse report showing employees by DEPTNO and JOB:

```
select deptno dno, job,
       max(case when deptno=10
                then ename else null end) as d10,
       max(case when deptno=20
                then ename else null end) as d20,
       max(case when deptno=30
                then ename else null end) as d30,
       max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as anals,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from (
Select deptno,
       job,
       ename,
       row_number()over(partition by job order by ename) rn_job,
       row_number()over(partition by job order by ename) rn_deptno
  from emp
       ) x
 group by deptno, job, rn_deptno, rn_job
 order by 1

DNO JOB       D10    D20   D30    CLERKS ANALS MGRS  PREZ SALES
--- --------- ------ ----- ------ ------ ----- ----- ---- ------
 10 CLERK     MILLER              MILLER
 10 MANAGER   CLARK                             CLARK
 10 PRESIDENT KING                                    KING
 20 ANALYST          FORD                FORD
 20 ANALYST          SCOTT               SCOTT
 20 CLERK            ADAMS        ADAMS
 20 CLERK            SMITH        SMITH
 20 MANAGER          JONES                     JONES
 30 CLERK                  JAMES  JAMES
 30 MANAGER                BLAKE             BLAKE
 30 SALESMAN               ALLEN                         ALLEN
 30 SALESMAN               MARTIN                        MARTIN
 30 SALESMAN               TURNER                        TURNER
 30 SALESMAN               WARD                          WARD
```

By simply modifying what you group by (hence the nonaggregate items in the previous SELECT list), you can produce reports with different formats. It is worth the time

of changing things around to understand how these formats change based on what you include in your GROUP BY clause.

12.3 Reverse Pivoting a Result Set

Problem

You want to transform columns to rows. Consider the following result set:

```
DEPTNO_10  DEPTNO_20  DEPTNO_30
---------- ---------- ----------
         3          5          6
```

You would like to convert that to the following:

```
DEPTNO COUNTS_BY_DEPT
------ --------------
    10              3
    20              5
    30              6
```

Some readers may have noticed that the first listing is the output from the first recipe in this chapter. To make this output available for this recipe, we can store it in a view with the following query:

```
create view emp_cnts as
(
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
          sum(case when deptno=20 then 1 else 0 end) as deptno_20,
          sum(case when deptno=30 then 1 else 0 end) as deptno_30
     from emp
)
```

In the solution and discussion that follow, the queries will refer to the EMP_CNTS view created by the preceding query.

Solution

Examining the desired result set, it's easy to see that you can execute a simple COUNT and GROUP BY on table EMP to produce the desired result. The object here, though, is to imagine that the data is not stored as rows; perhaps the data is denormalized and aggregated values are stored as multiple columns.

To convert columns to rows, use a Cartesian product. You'll need to know in advance how many columns you want to convert to rows because the table expression you use to create the Cartesian product must have a cardinality of at least the number of columns you want to transpose.

Rather than create a denormalized table of data, the solution for this recipe will use the solution from the first recipe of this chapter to create a "wide" result set. The full solution is as follows:

```
select dept.deptno,
       case dept.deptno
            when 10 then emp_cnts.deptno_10
            when 20 then emp_cnts.deptno_20
            when 30 then emp_cnts.deptno_30
       end as counts_by_dept
  from emp_cnts cross join
       (select deptno from dept where deptno <= 30) dept
```

Discussion

The view EMP_CNTS represents the denormalized view, or "wide" result set that you want to convert to rows, and is shown here:

```
DEPTNO_10   DEPTNO_20   DEPTNO_30
---------  ----------  ----------
        3           5           6
```

Because there are three columns, you will create three rows. Begin by creating a Cartesian product between inline view EMP_CNTS and some table expression that has at least three rows. The following code uses table DEPT to create the Cartesian product; DEPT has four rows:

```
select dept.deptno,
        emp_cnts.deptno_10,
        emp_cnts.deptno_20,
        emp_cnts.deptno_30
   from (
  Select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
         sum(case when deptno=20 then 1 else 0 end) as deptno_20,
         sum(case when deptno=30 then 1 else 0 end) as deptno_30
   from emp
        ) emp_cnts,
        (select deptno from dept where deptno <= 30) dept

 DEPTNO DEPTNO_10  DEPTNO_20  DEPTNO_30
 ------ ---------- ---------- ---------
     10          3          5         6
     20          3          5         6
     30          3          5         6
```

The Cartesian product enables you to return a row for each column in inline view EMP_CNTS. Since the final result set should have only the DEPTNO and the number of employees in said DEPTNO, use a CASE expression to transform the three columns into one:

```
 select dept.deptno,
        case dept.deptno
             when 10 then emp_cnts.deptno_10
             when 20 then emp_cnts.deptno_20
             when 30 then emp_cnts.deptno_30
        end as counts_by_dept
    from (
        emp_cnts
cross join (select deptno from dept where deptno <= 30) dept

  DEPTNO COUNTS_BY_DEPT
  ------ --------------
      10              3
      20              5
      30              6
```

12.4 Reverse Pivoting a Result Set into One Column

Problem

You want to return all columns from a query as just one column. For example, you want to return the ENAME, JOB, and SAL of all employees in DEPTNO 10, and you want to return all three values in one column. You want to return three rows for each employee and one row of white space between employees. You want to return the following result set:

```
EMPS
----------
CLARK
MANAGER
2450

KING
PRESIDENT
5000

MILLER
CLERK
1300
```

Solution

The key is to use a recursive CTE combined with Cartesian product to return four rows for each employee. Chapter 10 covers the recursive CTE we need, and it's explored further in Appendix B. Using the Cartesian join lets you return one column value per row and have an extra row for spacing between employees.

Use the window function ROW_NUMBER OVER to rank each row based on EMPNO (1–4). Then use a CASE expression to transform three columns into one (the keyword RECURSIVE is needed after the first WITH in PostgreSQL and MySQL):

```
with four_rows (id)
  as
(
 select 1
   union all
 select id+1
   from four_rows
   where id < 4
 )
 ,
  x_tab (ename,job,sal,rn )
   as
 (
  select  e.ename,e.job,e.sal,
    row_number()over(partition by e.empno
    order by e.empno)
    from emp e
    join four_rows on 1=1
  )

 select
   case rn
   when 1 then ename
   when 2 then job
   when 3 then cast(sal as char(4))
  end emps
from x_tab
```

Discussion

The first step is to use the window function ROW_NUMBER OVER to create a ranking for each employee in DEPTNO 10:

```
select e.ename,e.job,e.sal,
       row_number()over(partition by e.empno
                  order by e.empno) rn
from emp e
 where e.deptno=10

ENAME       JOB             SAL         RN
---------- --------- ---------- ----------
CLARK      MANAGER         2450          1
KING       PRESIDENT       5000          1
MILLER     CLERK           1300          1
```

At this point, the ranking doesn't mean much. You are partitioning by EMPNO, so the rank is 1 for all three rows in DEPTNO 10. Once you add the Cartesian product, the rank will begin to take shape, as shown in the following results:

```
with four_rows (id)
  as
  (select 1
  union all
  select id+1
  from four_rows
  where id < 4
  )
 select e.ename,e.job,e.sal,
 row_number()over(partition by e.empno
 order by e.empno)
  from emp e
  join four_rows on 1=1

  ENAME      JOB              SAL         RN
  ---------- --------- ---------- ----------
  CLARK      MANAGER         2450          1
  CLARK      MANAGER         2450          2
  CLARK      MANAGER         2450          3
  CLARK      MANAGER         2450          4
  KING       PRESIDENT       5000          1
  KING       PRESIDENT       5000          2
  KING       PRESIDENT       5000          3
  KING       PRESIDENT       5000          4
  MILLER     CLERK           1300          1
  MILLER     CLERK           1300          2
  MILLER     CLERK           1300          3
  MILLER     CLERK           1300          4
```

You should stop at this point and understand two key points:

- RN is no longer 1 for each employee; it is now a repeating sequence of values from 1 to 4, the reason being that window functions are applied after the FROM and WHERE clauses are evaluated. So, partitioning by EMPNO causes the RN to reset to 1 when a new employee is encountered.
- We've used a recursive CTE to ensure that for each employee there are four rows. We don't need the RECURSIVE keyword in SQL Server or DB2, but we do for Oracle, MySQL, and PostgreSQL.

The hard work is now done, and all that is left is to use a CASE expression to put ENAME, JOB, and SAL into one column for each employee (you need to use CAST to convert SAL to a string to keep CASE happy):

```
 with four_rows (id)
  as
  (select 1
```

```
 union all
 select id+1
 from four_rows
 where id < 4
 )
 ,
 x_tab (ename,job,sal,rn )
 as
 (select e.ename,e.job,e.sal,
row_number()over(partition by e.empno
order by e.empno)
 from emp e
 join four_rows on 1=1)

  select case rn
 when 1 then ename
 when 2 then job
 when 3 then cast(sal as char(4))
end emps
 from x_tab

 EMPS
 ----------
 CLARK
 MANAGER
 2450

 KING
 PRESIDENT
 5000

 MILLER
 CLERK
 1300
```

12.5 Suppressing Repeating Values from a Result Set

Problem

You are generating a report, and when two rows have the same value in a column, you want to display that value only once. For example, you want to return DEPTNO and ENAME from table EMP, you want to group all rows for each DEPTNO, and you want to display each DEPTNO only one time. You want to return the following result set:

```
DEPTNO ENAME
------ ---------
    10 CLARK
       KING
       MILLER
```

```
    20 SMITH
       ADAMS
       FORD
       SCOTT
       JONES
    30 ALLEN
       BLAKE
       MARTIN
       JAMES
       TURNER
       WARD
```

Solution

This is a simple formatting problem that is easily solved by the window function LAG OVER:

```
1   select
2          case when
3             lag(deptno)over(order by deptno) = deptno then null
4             else deptno end DEPTNO
5       , ename
6    from emp
```

Oracle users can also use DECODE as an alternative to CASE:

```
1 select to_number(
2          decode(lag(deptno)over(order by deptno),
3                 deptno,null,deptno)
4        ) deptno, ename
5   from emp
```

Discussion

The first step is to use the window function LAG OVER to return the prior DEPTNO for each row:

```
select lag(deptno)over(order by deptno) lag_deptno,
       deptno,
       ename
  from emp

LAG_DEPTNO     DEPTNO ENAME
---------- ---------- ----------
                   10 CLARK
        10         10 KING
        10         10 MILLER
        10         20 SMITH
        20         20 ADAMS
        20         20 FORD
        20         20 SCOTT
        20         20 JONES
```

```
        20         30 ALLEN
        30         30 BLAKE
        30         30 MARTIN
        30         30 JAMES
        30         30 TURNER
        30         30 WARD
```

If you inspect the previous result set, you can easily see where DEPTNO matches LAG_ DEPTNO. For those rows, you want to set DEPTNO to NULL. Do that by using DECODE (TO_NUMBER is included to cast DEPTNO as a number):

```
select to_number(
          CASE WHEN (lag(deptno)over(order by deptno)
= deptno THEN null else deptno END deptno ,
                 deptno,null,deptno)
       ) deptno, ename
  from emp

DEPTNO ENAME
------ ----------
    10 CLARK
       KING
       MILLER
    20 SMITH
       ADAMS
       FORD
       SCOTT
       JONES
    30 ALLEN
       BLAKE
       MARTIN
       JAMES
       TURNER
       WARD
```

12.6 Pivoting a Result Set to Facilitate Inter-Row Calculations

Problem

You want to make calculations involving data from multiple rows. To make your job easier, you want to pivot those rows into columns such that all values you need are then in a single row.

In this book's example data, DEPTNO 20 is the department with the highest combined salary, which you can confirm by executing the following query:

```
select deptno, sum(sal) as sal
  from emp
 group by deptno
```

```
DEPTNO        SAL
------ ----------
    10       8750
    20      10875
    30       9400
```

You want to calculate the difference between the salaries of DEPTNO 20 and DEPTNO 10 and between DEPTNO 20 and DEPTNO 30.

The final result will look like this:

```
d20_10_diff    d20_30_diff
------------   ----------
2125            1475
```

Solution

Transpose the totals using the aggregate function SUM and a CASE expression. Then code your expressions in the select list:

```
1 select d20_sal - d10_sal as d20_10_diff,
2        d20_sal - d30_sal as d20_30_diff
3   from (
4 select sum(case when deptno=10 then sal end) as d10_sal,
5        sum(case when deptno=20 then sal end) as d20_sal,
6        sum(case when deptno=30 then sal end) as d30_sal
7   from emp
8        ) totals_by_dept
```

It is also possible to write this query using a CTE, which some people may find more readable:

```
with totals_by_dept (d10_sal, d20_sal, d30_sal)
as
(select
          sum(case when deptno=10 then sal end) as d10_sal,
           sum(case when deptno=20 then sal end) as d20_sal,
           sum(case when deptno=30 then sal end) as d30_sal

from emp)

select   d20_sal - d10_sal as d20_10_diff,
          d20_sal - d30_sal as d20_30_diff
     from totals_by_dept
```

Discussion

The first step is to pivot the salaries for each DEPTNO from rows to columns by using a CASE expression:

```
select case when deptno=10 then sal end as d10_sal,
       case when deptno=20 then sal end as d20_sal,
       case when deptno=30 then sal end as d30_sal
  from emp

D10_SAL    D20_SAL    D30_SAL
------- ---------- ----------
               800
                          1600
                          1250
             2975
                          1250
                          2850
   2450
             3000
   5000
                          1500
             1100
                           950
             3000
   1300
```

The next step is to sum all the salaries for each DEPTNO by applying the aggregate function SUM to each CASE expression:

```
select sum(case when deptno=10 then sal end) as d10_sal,
       sum(case when deptno=20 then sal end) as d20_sal,
       sum(case when deptno=30 then sal end) as d30_sal
  from emp

D10_SAL    D20_SAL    D30_SAL
------- ---------- ----------
   8750      10875       9400
```

The final step is to simply wrap the previous SQL in an inline view and perform the subtractions.

12.7 Creating Buckets of Data, of a Fixed Size

Problem

You want to organize data into evenly sized buckets, with a predetermined number of elements in each bucket. The total number of buckets may be unknown, but you want to ensure that each bucket has five elements. For example, you want to organize the employees in table EMP into groups of five based on the value of EMPNO, as shown in the following results:

```
GRP      EMPNO ENAME
--- ---------- -------
  1       7369 SMITH
  1       7499 ALLEN
```

```
  1       7521 WARD
  1       7566 JONES
  1       7654 MARTIN
  2       7698 BLAKE
  2       7782 CLARK
  2       7788 SCOTT
  2       7839 KING
  2       7844 TURNER
  3       7876 ADAMS
  3       7900 JAMES
  3       7902 FORD
  3       7934 MILLER
```

Solution

The solution to this problem is greatly simplified by functions for ranking rows. Once the rows are ranked, creating buckets of five is simply a matter of dividing and then taking the mathematical ceiling of the quotient.

Use the window function ROW_NUMBER OVER to rank each employee by EMPNO. Then divide by five to create the groups (SQL Server users will use CEILING, not CEIL):

```
1 select ceil(row_number()over(order by empno)/5.0) grp,
2        empno,
3        ename
4   from emp
```

Discussion

The window function ROW_NUMBER OVER assigns a rank or "row number" to each row sorted by EMPNO:

```
select row_number()over(order by empno) rn,
       empno,
       ename
  from emp

RN      EMPNO ENAME
-- ---------- ----------
 1       7369 SMITH
 2       7499 ALLEN
 3       7521 WARD
 4       7566 JONES
 5       7654 MARTIN
 6       7698 BLAKE
 7       7782 CLARK
 8       7788 SCOTT
 9       7839 KING
10       7844 TURNER
11       7876 ADAMS
```

```
12       7900 JAMES
13       7902 FORD
14       7934 MILLER
```

The next step is to apply the function CEIL (or CEILING) after dividing ROW_NUMBER OVER by five. Dividing by five logically organizes the rows into groups of five (i.e., five values less than or equal to 1, five values greater than 1 but less than or equal to 2); the remaining group (composed of the last 4 rows since 14, the number of rows in table EMP, is not a multiple of 5) has a value greater than 2 but less than or equal to 3.

The CEIL function will return the smallest whole number greater than the value passed to it; this will create whole number groups. The results of the division and application of the CEIL are shown here. You can follow the order of operation from left to right, from RN to DIVISION to GRP:

```
select row_number()over(order by empno) rn,
       row_number()over(order by empno)/5.0 division,
       ceil(row_number()over(order by empno)/5.0) grp,
       empno,
       ename
  from emp

RN   DIVISION GRP EMPNO ENAME
-- ---------- --- ----- ----------
 1         .2   1  7369 SMITH
 2         .4   1  7499 ALLEN
 3         .6   1  7521 WARD
 4         .8   1  7566 JONES
 5          1   1  7654 MARTIN
 6        1.2   2  7698 BLAKE
 7        1.4   2  7782 CLARK
 8        1.6   2  7788 SCOTT
 9        1.8   2  7839 KING
10          2   2  7844 TURNER
11        2.2   3  7876 ADAMS
12        2.4   3  7900 JAMES
13        2.6   3  7902 FORD
14        2.8   3  7934 MILLER
```

12.8 Creating a Predefined Number of Buckets

Problem

You want to organize your data into a fixed number of buckets. For example, you want to organize the employees in table EMP into four buckets. The result set should look similar to the following:

```
GRP EMPNO ENAME
--- ----- ---------
  1  7369 SMITH
  1  7499 ALLEN
  1  7521 WARD
  1  7566 JONES
  2  7654 MARTIN
  2  7698 BLAKE
  2  7782 CLARK
  2  7788 SCOTT
  3  7839 KING
  3  7844 TURNER
  3  7876 ADAMS
  4  7900 JAMES
  4  7902 FORD
  4  7934 MILLER
```

This is a common way to organize categorical data as dividing a set into a number of smaller equal sized sets is an important first step for many kinds of analysis. For example, taking the averages of these groups on salary or any other value may reveal a trend that is concealed by variability when looking at the cases individually.

This problem is the opposite of the previous recipe, where you had an unknown number of buckets but a predetermined number of elements in each bucket. In this recipe, the goal is such that you may not necessarily know how many elements are in each bucket, but you are defining a fixed (known) number of buckets to be created.

Solution

The solution to this problem is simple now that the NTILE function is widely available. NTILE organizes an ordered set into the number of buckets you specify, with any stragglers distributed into the available buckets starting from the first bucket. The desired result set for this recipe reflects this: buckets 1 and 2 have four rows, while buckets 3 and 4 have three rows.

Use the NTILE window function to create four buckets:

```
1 select ntile(4)over(order by empno) grp,
2        empno,
3        ename
4   from emp
```

Discussion

All the work is done by the NTILE function. The ORDER BY clause puts the rows into the desired order, and the function itself then assigns a group number to each row, for example, so that the first quarter (in this case) are put into group one, the second into group two, etc.

12.9 Creating Horizontal Histograms

Problem

You want to use SQL to generate histograms that extend horizontally. For example, you want to display the number of employees in each department as a horizontal histogram with each employee represented by an instance of *. You want to return the following result set:

```
DEPTNO CNT
------ ----------
    10 ***
    20 *****
    30 ******
```

Solution

The key to this solution is to use the aggregate function COUNT and use GROUP BY DEPTNO to determine the number of employees in each DEPTNO. The value returned by COUNT is then passed to a string function that generates a series of * characters.

DB2

Use the REPEAT function to generate the histogram:

```
1 select deptno,
2        repeat('*',count(*)) cnt
3   from emp
4  group by deptno
```

Oracle, PostgreSQL, and MySQL

Use the LPAD function to generate the needed strings of * characters:

```
1 select deptno,
2        lpad('*',count(*),'*') as cnt
3   from emp
4  group by deptno
```

SQL Server

Generate the histogram using the REPLICATE function:

```
1 select deptno,
2        replicate('*',count(*)) cnt
3   from emp
4  group by deptno
```

Discussion

The technique is the same for all vendors. The only difference lies in the string function used to return a * for each employee. The Oracle solution will be used for this discussion, but the explanation is relevant for all the solutions.

The first step is to count the number of employees in each department:

```
select deptno,
       count(*)
  from emp
 group by deptno

DEPTNO   COUNT(*)
------ ----------
    10          3
    20          5
    30          6
```

The next step is to use the value returned by COUNT to control the number of * characters to return for each department. Simply pass COUNT(*) as an argument to the string function LPAD to return the desired number of *:

```
select deptno,
       lpad('*',count(*),'*') as cnt
  from emp
 group by deptno

DEPTNO CNT
------ ----------
    10 ***
    20 *****
    30 ******
```

For PostgreSQL users, you may need to use CAST to ensure that COUNT(*) returns an integer as shown here:

```
select deptno,
       lpad('*',count(*)::integer,'*') as cnt
  from emp
 group by deptno

DEPTNO CNT
------ ----------
    10 ***
    20 *****
    30 ******
```

This CAST is necessary because PostgreSQL requires the numeric argument to LPAD to be an integer.

12.10 Creating Vertical Histograms

Problem

You want to generate a histogram that grows from the bottom up. For example, you want to display the number of employees in each department as a vertical histogram with each employee represented by an instance of *. You want to return the following result set:

```
D10 D20 D30
--- --- ---
        *
    *   *
    *   *
*   *   *
*   *   *
*   *   *
```

Solution

The technique used to solve this problem is built on a technique used earlier in this chapter: use the ROW_NUMBER OVER function to uniquely identify each instance of * for each DEPTNO. Use the aggregate function MAX to pivot the result set and group by the values returned by ROW_NUMBER OVER (SQL Server users should not use DESC in the ORDER BY clause):

```
 1 select max(deptno_10) d10,
 2        max(deptno_20) d20,
 3        max(deptno_30) d30
 4   from (
 5 select row_number()over(partition by deptno order by empno) rn,
 6        case when deptno=10 then '*' else null end deptno_10,
 7        case when deptno=20 then '*' else null end deptno_20,
 8        case when deptno=30 then '*' else null end deptno_30
 9   from emp
10        ) x
11  group by rn
12  order by 1 desc, 2 desc, 3 desc
```

Discussion

The first step is to use the window function ROW_NUMBER to uniquely identify each instance of * in each department. Use a CASE expression to return a * for each employee in each department:

```
select row_number()over(partition by deptno order by empno) rn,
       case when deptno=10 then '*' else null end deptno_10,
       case when deptno=20 then '*' else null end deptno_20,
       case when deptno=30 then '*' else null end deptno_30
  from emp

RN DEPTNO_10  DEPTNO_20  DEPTNO_30
-- ---------- ---------- ---------
 1 *
 2 *
 3 *
 1            *
 2            *
 3            *
 4            *
 5            *
 1                       *
 2                       *
 3                       *
 4                       *
 5                       *
 6                       *
```

The next and last step is to use the aggregate function MAX on each CASE expression, grouping by RN to remove the NULLs from the result set. Order the results ASC or DESC depending on how your RDBMS sorts NULLs:

```
select max(deptno_10) d10,
       max(deptno_20) d20,
       max(deptno_30) d30
  from (
select row_number()over(partition by deptno order by empno) rn,
       case when deptno=10 then '*' else null end deptno_10,
       case when deptno=20 then '*' else null end deptno_20,
       case when deptno=30 then '*' else null end deptno_30
  from emp
       ) x
 group by rn
 order by 1 desc, 2 desc, 3 desc

D10 D20 D30
--- --- ---
        *
    *   *
    *   *
*   *   *
*   *   *
*   *   *
```

12.11 Returning Non-GROUP BY Columns

Problem

You are executing a GROUP BY query, and you want to return columns in your select list that are not also listed in your GROUP BY clause. This is not normally possible, as such ungrouped columns would not represent a single value per row.

Say that you want to find the employees who earn the highest and lowest salaries in each department, as well as the employees who earn the highest and lowest salaries in each job. You want to see each employee's name, the department he works in, his job title, and his salary. You want to return the following result set:

```
DEPTNO ENAME  JOB         SAL DEPT_STATUS     JOB_STATUS
------ ------ --------- ----- --------------- --------------
    10 MILLER CLERK      1300 LOW SAL IN DEPT TOP SAL IN JOB
    10 CLARK  MANAGER    2450                 LOW SAL IN JOB
    10 KING   PRESIDENT  5000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SCOTT  ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 FORD   ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SMITH  CLERK       800 LOW SAL IN DEPT LOW SAL IN JOB
    20 JONES  MANAGER    2975                 TOP SAL IN JOB
    30 JAMES  CLERK       950 LOW SAL IN DEPT
    30 MARTIN SALESMAN   1250                 LOW SAL IN JOB
    30 WARD   SALESMAN   1250                 LOW SAL IN JOB
    30 ALLEN  SALESMAN   1600                 TOP SAL IN JOB
    30 BLAKE  MANAGER    2850 TOP SAL IN DEPT
```

Unfortunately, including all these columns in the SELECT clause will ruin the grouping. Consider the following example: employee KING earns the highest salary. You want to verify this with the following query:

```
select ename,max(sal)
  from empgroup by ename
```

Instead of seeing KING and KING's salary, the previous query will return all 14 rows from table EMP. The reason is because of the grouping: the MAX(SAL) is applied to each ENAME. So, it would seem the previous query can be stated as "find the employee with the highest salary," but in fact what it is doing is "find the highest salary for each ENAME in table EMP." This recipe explains a technique for including ENAME without the need to GROUP BY that column.

Solution

Use an inline view to find the high and low salaries by DEPTNO and JOB. Then keep only the employees who make those salaries.

Use the window functions MAX OVER and MIN OVER to find the highest and lowest salaries by DEPTNO and JOB. Then keep the rows where the salaries are those that are highest or lowest by DEPTNO or JOB:

```
 1 select deptno,ename,job,sal,
 2        case when sal = max_by_dept
 3             then 'TOP SAL IN DEPT'
 4             when sal = min_by_dept
 5             then 'LOW SAL IN DEPT'
 6        end dept_status,
 7        case when sal = max_by_job
 8             then 'TOP SAL IN JOB'
 9             when sal = min_by_job
10             then 'LOW SAL IN JOB'
11        end job_status
12   from (
13 select deptno,ename,job,sal,
14        max(sal)over(partition by deptno) max_by_dept,
15        max(sal)over(partition by job)   max_by_job,
16        min(sal)over(partition by deptno) min_by_dept,
17        min(sal)over(partition by job)   min_by_job
18   from emp
19        ) emp_sals
20  where sal in (max_by_dept,max_by_job,
21                min_by_dept,min_by_job)
```

Discussion

The first step is to use the window functions MAX OVER and MIN OVER to find the highest and lowest salaries by DEPTNO and JOB:

```
select deptno,ename,job,sal,
       max(sal)over(partition by deptno) maxDEPT,
       max(sal)over(partition by job) maxJOB,
       min(sal)over(partition by deptno) minDEPT,
       min(sal)over(partition by job) minJOB
  from emp

DEPTNO ENAME  JOB         SAL MAXDEPT MAXJOB MINDEPT MINJOB
------ ------ --------- ----- ------- ------ ------- ------
    10 MILLER CLERK      1300    5000   1300    1300    800
    10 CLARK  MANAGER    2450    5000   2975    1300   2450
    10 KING   PRESIDENT  5000    5000   5000    1300   5000
    20 SCOTT  ANALYST    3000    3000   3000     800   3000
    20 FORD   ANALYST    3000    3000   3000     800   3000
    20 SMITH  CLERK       800    3000   1300     800    800
    20 JONES  MANAGER    2975    3000   2975     800   2450
    20 ADAMS  CLERK      1100    3000   1300     800    800
    30 JAMES  CLERK       950    2850   1300     950    800
    30 MARTIN SALESMAN   1250    2850   1600     950   1250
    30 TURNER SALESMAN   1500    2850   1600     950   1250
    30 WARD   SALESMAN   1250    2850   1600     950   1250
```

```
    30 ALLEN  SALESMAN   1600    2850   1600     950   1250
    30 BLAKE  MANAGER    2850    2850   2975     950   2450
```

At this point, every salary can be compared with the highest and lowest salaries by DEPTNO and JOB. Notice that the grouping (the inclusion of multiple columns in the SELECT clause) does not affect the values returned by MIN OVER and MAX OVER. This is the beauty of window functions: the aggregate is computed over a defined "group" or partition and returns multiple rows for each group. The last step is to simply wrap the window functions in an inline view and keep only those rows that match the values returned by the window functions. Use a simple CASE expression to display the "status" of each employee in the final result set:

```
select deptno,ename,job,sal,
       case when sal = max_by_dept
            then 'TOP SAL IN DEPT'
            when sal = min_by_dept
            then 'LOW SAL IN DEPT'
       end dept_status,
       case when sal = max_by_job
            then 'TOP SAL IN JOB'
            when sal = min_by_job
            then 'LOW SAL IN JOB'
       end job_status
  from (
select deptno,ename,job,sal,
       max(sal)over(partition by deptno) max_by_dept,
       max(sal)over(partition by job) max_by_job,
       min(sal)over(partition by deptno) min_by_dept,
       min(sal)over(partition by job) min_by_job
  from emp
       ) x
 where sal in (max_by_dept,max_by_job,
               min_by_dept,min_by_job)

DEPTNO ENAME  JOB         SAL DEPT_STATUS     JOB_STATUS
------ ------ --------- ----- --------------- --------------
    10 MILLER CLERK      1300 LOW SAL IN DEPT TOP SAL IN JOB
    10 CLARK  MANAGER    2450                 LOW SAL IN JOB
    10 KING   PRESIDENT  5000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SCOTT  ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 FORD   ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SMITH  CLERK       800 LOW SAL IN DEPT LOW SAL IN JOB
    20 JONES  MANAGER    2975                 TOP SAL IN JOB
    30 JAMES  CLERK       950 LOW SAL IN DEPT
    30 MARTIN SALESMAN   1250                 LOW SAL IN JOB
    30 WARD   SALESMAN   1250                 LOW SAL IN JOB
    30 ALLEN  SALESMAN   1600                 TOP SAL IN JOB
    30 BLAKE  MANAGER    2850 TOP SAL IN DEPT
```

12.12 Calculating Simple Subtotals

Problem

For the purposes of this recipe, a *simple subtotal* is defined as a result set that contains values from the aggregation of one column along with a grand total value for the table. An example would be a result set that sums the salaries in table EMP by JOB and that also includes the sum of all salaries in table EMP. The summed salaries by JOB are the subtotals, and the sum of all salaries in table EMP is the grand total. Such a result set should look as follows:

```
JOB              SAL
--------- ----------
ANALYST         6000
CLERK           4150
MANAGER         8275
PRESIDENT       5000
SALESMAN        5600
TOTAL          29025
```

Solution

The ROLLUP extension to the GROUP BY clause solves this problem perfectly. If ROLLUP is not available for your RDBMS, you can solve the problem, albeit with more difficulty, using a scalar subquery or a UNION query.

DB2 and Oracle

Use the aggregate function SUM to sum the salaries, and use the ROLLUP extension of GROUP BY to organize the results into subtotals (by JOB) and a grand total (for the whole table):

```
1 select case grouping(job)
2            when 0 then job
3            else 'TOTAL'
4        end job,
5        sum(sal) sal
6   from emp
7  group by rollup(job)
```

SQL Server and MySQL

Use the aggregate function SUM to sum the salaries, and use WITH ROLLUP to organize the results into subtotals (by JOB) and a grand total (for the whole table). Then use COALESCE to supply the label TOTAL for the grand total row (which will otherwise have a NULL in the JOB column):

```
1 select coalesce(job,'TOTAL') job,
2        sum(sal) sal
3   from emp
4  group by job with rollup
```

With SQL Server, you also have the option to use the GROUPING function shown in the Oracle/DB2 recipe rather than COALESCE to determine the level of aggregation.

PostgreSQL

Similar to the SQL Server and MySQL solutions, you use the ROLLUP extension to GROUP BY with slightly different syntax:

```
select coalesce(job,'TOTAL') job,
        sum(sal) sal
   from emp
  group by rollup(job)
```

Discussion

DB2 and Oracle

The first step is to use the aggregate function SUM, grouping by JOB in order to sum the salaries by JOB:

```
select job, sum(sal) sal
  from emp
 group by job

JOB         SAL
--------- -----
ANALYST    6000
CLERK      4150
MANAGER    8275
PRESIDENT  5000
SALESMAN   5600
```

The next step is to use the ROLLUP extension to GROUP BY to produce a grand total for all salaries along with the subtotals for each JOB:

```
select job, sum(sal) sal
  from emp
 group by rollup(job)

JOB           SAL
--------- -------
ANALYST      6000
CLERK        4150
MANAGER      8275
PRESIDENT    5000
SALESMAN     5600
            29025
```

The last step is to use the GROUPING function in the JOB column to display a label for the grand total. If the value of JOB is NULL, the GROUPING function will return 1, which signifies that the value for SAL is the grand total created by ROLLUP. If the value of JOB is not NULL, the GROUPING function will return 0, which signifies the value for SAL is the result of the GROUP BY, not the ROLLUP. Wrap the call to GROUPING(JOB) in a CASE expression that returns either the job name or the label TOTAL, as appropriate:

```
select case grouping(job)
            when 0 then job
            else 'TOTAL'
       end job,
       sum(sal) sal
  from emp
 group by rollup(job)
```

```
JOB              SAL
--------- ----------
ANALYST         6000
CLERK           4150
MANAGER         8275
PRESIDENT       5000
SALESMAN        5600
TOTAL          29025
```

SQL Server and MySQL

The first step is to use the aggregate function SUM, grouping the results by JOB to generate salary sums by JOB:

```
select job, sum(sal) sal
  from emp
 group by job
```

```
JOB         SAL
--------- -----
ANALYST    6000
CLERK      4150
MANAGER    8275
PRESIDENT  5000
SALESMAN   5600
```

The next step is to use GROUP BY's ROLLUP extension to produce a grand total for all salaries along with the subtotals for each JOB:

```
select job, sum(sal) sal
  from emp
 group by job with rollup
```

```
JOB           SAL
--------- -------
ANALYST      6000
CLERK        4150
MANAGER      8275
PRESIDENT    5000
SALESMAN     5600
            29025
```

The last step is to use the COEALESCE function against the JOB column. If the value of JOB is NULL, the value for SAL is the grand total created by ROLLUP. If the value of JOB is not NULL, the value for SAL is the result of the "regular" GROUP BY, not the ROLLUP:

```
select coalesce(job,'TOTAL') job,
       sum(sal) sal
  from emp
 group by job with rollup

JOB              SAL
--------- ----------
ANALYST         6000
CLERK           4150
MANAGER         8275
PRESIDENT       5000
SALESMAN        5600
TOTAL          29025
```

PostgreSQL

The solution is the same in its manner of operation as the preceeding solution for MySQL and SQL Server. The only difference is the syntax for the ROLLUP clause: write ROLLUP(JOB) after GROUP BY.

12.13 Calculating Subtotals for All Possible Expression Combinations

Problem

You want to find the sum of all salaries by DEPTNO, and by JOB, for every JOB/DEPTNO combination. You also want a grand total for all salaries in table EMP. You want to return the following result set:

```
DEPTNO JOB       CATEGORY                  SAL
------ --------- --------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
```

```
30 CLERK     TOTAL BY DEPT AND JOB     950
30 SALESMAN  TOTAL BY DEPT AND JOB    5600
30 MANAGER   TOTAL BY DEPT AND JOB    2850
20 MANAGER   TOTAL BY DEPT AND JOB    2975
20 ANALYST   TOTAL BY DEPT AND JOB    6000
   CLERK     TOTAL BY JOB             4150
   ANALYST   TOTAL BY JOB             6000
   MANAGER   TOTAL BY JOB             8275
   PRESIDENT TOTAL BY JOB             5000
   SALESMAN  TOTAL BY JOB             5600
10           TOTAL BY DEPT            8750
30           TOTAL BY DEPT            9400
20           TOTAL BY DEPT           10875
             GRAND TOTAL FOR TABLE   29025
```

Solution

Extensions added to GROUP BY in recent years make this a fairly easy problem to solve. If your platform does not supply such extensions for computing various levels of subtotals, then you must compute them manually (via self-joins or scalar subqueries).

DB2

For DB2, you will need to use CAST to return from GROUPING as the CHAR(1) data type:

```
select deptno,
       job,
       case cast(grouping(deptno) as char(1))||
            cast(grouping(job) as char(1))
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'TOTAL FOR TABLE'
       end category,
       sum(sal)
  from emp
group by cube(deptno,job)
order by grouping(job),grouping(deptno)
```

Oracle

Use the CUBE extension to the GROUP BY clause with the concatenation operator ||:

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
```

```
 7            when '11' then 'GRAND TOTALFOR TABLE'
 8       end category,
 9       sum(sal) sal
10   from emp
11  group by cube(deptno,job)
12  order by grouping(job),grouping(deptno)
```

SQL Server

Use the CUBE extension to the GROUP BY clause. For SQL Server, you will need to CAST the results from GROUPING to CHAR(1), and you will need to use the + operator for concatenation (as opposed to Oracle's || operator):

```
 1 select deptno,
 2        job,
 3        case cast(grouping(deptno)as char(1))+
 4             cast(grouping(job)as char(1))
 5             when '00' then 'TOTAL BY DEPT AND JOB'
 6             when '10' then 'TOTAL BY JOB'
 7             when '01' then 'TOTAL BY DEPT'
 8             when '11' then 'GRAND TOTAL FOR TABLE'
 9        end category,
10        sum(sal) sal
11   from emp
12  group by deptno,job with cube
13  order by grouping(job),grouping(deptno)
```

PostgreSQL

PostgreSQL is similar to the preceding, but with slightly different syntax for the CUBE operator and the concatenation:

```
select deptno,job
,case concat(
cast (grouping(deptno) as char(1)),cast (grouping(job) as char(1))
  )
  when '00' then 'TOTAL BY DEPT AND JOB'
               when '10' then 'TOTAL BY JOB'
               when '01' then 'TOTAL BY DEPT'
               when '11' then 'GRAND TOTAL FOR TABLE'
          end category
  , sum(sal) as sal
    from emp
    group by cube(deptno,job)
```

MySQL

Although part of the functionality is available, it is not complete, as MySQL has no CUBE function. Hence, use multiple UNION ALLs, creating different sums for each:

```
 1 select deptno, job,
 2        'TOTAL BY DEPT AND JOB' as category,
 3        sum(sal) as sal
 4   from emp
 5  group by deptno, job
 6  union all
 7 select null, job, 'TOTAL BY JOB', sum(sal)
 8   from emp
 9  group by job
10  union all
11 select deptno, null, 'TOTAL BY DEPT', sum(sal)
12   from emp
13  group by deptno
14  union all
15 select null,null,'GRAND TOTAL FOR TABLE', sum(sal)
16  from emp
```

Discussion

Oracle, DB2, and SQL Server

The solutions for all three are essentially the same. The first step is to use the aggregate function SUM and group by both DEPTNO and JOB to find the total salaries for each JOB and DEPTNO combination:

```
select deptno, job, sum(sal) sal
  from emp
 group by deptno, job

DEPTNO JOB           SAL
------ --------- -------
    10 CLERK        1300
    10 MANAGER      2450
    10 PRESIDENT    5000
    20 CLERK        1900
    20 ANALYST      6000
    20 MANAGER      2975
    30 CLERK         950
    30 MANAGER      2850
    30 SALESMAN     5600
```

The next step is to create subtotals by JOB and DEPTNO along with the grand total for the whole table. Use the CUBE extension to the GROUP BY clause to perform aggregations on SAL by DEPTNO, JOB, and for the whole table:

```
select deptno,
       job,
       sum(sal) sal
  from emp
 group by cube(deptno,job)
```

```
DEPTNO JOB           SAL
------ --------- -------
                   29025
       CLERK        4150
       ANALYST      6000
       MANAGER      8275
       SALESMAN     5600
       PRESIDENT    5000
    10              8750
    10 CLERK        1300
    10 MANAGER      2450
    10 PRESIDENT    5000
    20             10875
    20 CLERK        1900
    20 ANALYST      6000
    20 MANAGER      2975
    30              9400
    30 CLERK         950
    30 MANAGER      2850
    30 SALESMAN     5600
```

Next, use the GROUPING function in conjunction with CASE to format the results into more meaningful output. The value from GROUPING(JOB) will be 1 or 0 depending on whether the values for SAL are due to the GROUP BY or the CUBE. If the results are due to the CUBE, the value will be 1; otherwise, it will be 0. The same goes for GROUPING(DEPTNO). Looking at the first step of the solution, you should see that grouping is done by DEPTNO and JOB. Thus, the expected values from the calls to GROUPING when a row represents a combination of both DEPTNO and JOB is 0. The following query confirms this:

```
select deptno,
       job,
       grouping(deptno) is_deptno_subtotal,
       grouping(job) is_job_subtotal,
       sum(sal) sal
  from emp
 group by cube(deptno,job)
 order by 3,4

DEPTNO JOB       IS_DEPTNO_SUBTOTAL IS_JOB_SUBTOTAL     SAL
------ --------- ------------------ --------------- -------
    10 CLERK                     0               0    1300
    10 MANAGER                   0               0    2450
    10 PRESIDENT                 0               0    5000
    20 CLERK                     0               0    1900
    30 CLERK                     0               0     950
    30 SALESMAN                  0               0    5600
    30 MANAGER                   0               0    2850
    20 MANAGER                   0               0    2975
    20 ANALYST                   0               0    6000
    10                           0               1    8750
```

```
    20                          0              1   10875
    30                          0              1    9400
       CLERK                    1              0    4150
       ANALYST                  1              0    6000
       MANAGER                  1              0    8275
       PRESIDENT                1              0    5000
       SALESMAN                 1              0    5600
                                1              1   29025
```

The final step is to use a CASE expression to determine which category each row belongs to based on the values returned by GROUPING(JOB) and GROUPING(DEPTNO) concatenated:

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by cube(deptno,job)
 order by grouping(job),grouping(deptno)

DEPTNO JOB       CATEGORY                  SAL
------ --------- --------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
       CLERK     TOTAL BY JOB             4150
       ANALYST   TOTAL BY JOB             6000
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
    10           TOTAL BY DEPT            8750
    30           TOTAL BY DEPT            9400
    20           TOTAL BY DEPT           10875
                 GRAND TOTAL FOR TABLE   29025
```

This Oracle solution implicitly converts the results from the GROUPING functions to a character type in preparation for concatenating the two values. DB2 and SQL Server users will need to explicitly CAST the results of the GROUPING functions to CHAR(1), as shown in the solution. In addition, SQL Server users must use the +

operator, and not the || operator, to concatenate the results from the two GROUPING calls into one string.

For Oracle and DB2 users, there is an additional extension to GROUP BY called GROUPING SETS; this extension is extremely useful. For example, you can use GROUPING SETS to mimic the output created by CUBE as is shown here (DB2 and SQL Server users will need to use CAST to ensure the values returned by the GROUPING function are in the correct format in the same way as in the CUBE solution):

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((deptno),(job),(deptno,job),())

DEPTNO JOB       CATEGORY                  SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    30 CLERK     TOTAL BY DEPT AND JOB     950
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
       CLERK     TOTAL BY JOB             4150
       ANALYST   TOTAL BY JOB             6000
       MANAGER   TOTAL BY JOB             8275
       SALESMAN  TOTAL BY JOB             5600
       PRESIDENT TOTAL BY JOB             5000
    10           TOTAL BY DEPT            8750
    20           TOTAL BY DEPT           10875
    30           TOTAL BY DEPT            9400
                 GRAND TOTAL FOR TABLE   29025
```

What's great about GROUPING SETS is that it allows you to define the groups. The GROUPING SETS clause in the preceding query causes groups to be created by DEPTNO, by JOB, and by the combination of DEPTNO and JOB, and finally the empty parentheses requests a grand total. GROUPING SETS gives you enormous flexibility for creating reports with different levels of aggregation; for example, if you wanted to modify the preceding example to exclude the GRAND TOTAL, simply modify the GROUPING SETS clause by excluding the empty parentheses:

```
/* no grand total */

select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((deptno),(job),(deptno,job))


DEPTNO JOB       CATEGORY                    SAL
------ --------- -------------------- ----------
    10 CLERK     TOTAL BY DEPT AND JOB       1300
    20 CLERK     TOTAL BY DEPT AND JOB       1900
    30 CLERK     TOTAL BY DEPT AND JOB        950
    20 ANALYST   TOTAL BY DEPT AND JOB       6000
    10 MANAGER   TOTAL BY DEPT AND JOB       2450
    20 MANAGER   TOTAL BY DEPT AND JOB       2975
    30 MANAGER   TOTAL BY DEPT AND JOB       2850
    30 SALESMAN  TOTAL BY DEPT AND JOB       5600
    10 PRESIDENT TOTAL BY DEPT AND JOB       5000
       CLERK     TOTAL BY JOB                4150
       ANALYST   TOTAL BY JOB                6000
       MANAGER    TOTAL BY JOB                8275
       SALESMAN  TOTAL BY JOB                5600
       PRESIDENT TOTAL BY JOB                5000
    10           TOTAL BY DEPT               8750
    20           TOTAL BY DEPT              10875
    30           TOTAL BY DEPT               9400
```

You can also eliminate a subtotal, such as the one on DEPTNO, simply by omitting (DEPTNO) from the GROUPING SETS clause:

```
/* nosubtotals by DEPTNO */

select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((job),(deptno,job),())
 order by 3
```

```
DEPTNO JOB       CATEGORY                     SAL
------ --------- -------------------- ----------
                 GRAND TOTAL FOR TABLE      29025
    10 CLERK     TOTAL BY DEPT AND JOB       1300
    20 CLERK     TOTAL BY DEPT AND JOB       1900
    30 CLERK     TOTAL BY DEPT AND JOB        950
    20 ANALYST   TOTAL BY DEPT AND JOB       6000
    20 MANAGER   TOTAL BY DEPT AND JOB       2975
    30 MANAGER   TOTAL BY DEPT AND JOB       2850
    30 SALESMAN  TOTAL BY DEPT AND JOB       5600
    10 PRESIDENT TOTAL BY DEPT AND JOB       5000
    10 MANAGER   TOTAL BY DEPT AND JOB       2450
       CLERK     TOTAL BY JOB                4150
       SALESMAN  TOTAL BY JOB                5600
       PRESIDENT TOTAL BY JOB                5000
       MANAGER   TOTAL BY JOB                8275
       ANALYST   TOTAL BY JOB                6000
```

As you can see, GROUPING SETS makes it easy indeed to play around with totals and subtotals to look at your data from different angles.

MySQL

The first step is to use the aggregate function SUM and group by both DEPTNO and JOB:

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job

DEPTNO JOB       CATEGORY                   SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
```

The next step is to use UNION ALL to append TOTAL BY JOB sums:

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
```

```
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job

DEPTNO JOB       CATEGORY                  SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
       ANALYST   TOTAL BY JOB             6000
       CLERK     TOTAL BY JOB             4150
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
```

The next step is to UNION ALL the sum of all the salaries by DEPTNO:

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
 union all
select deptno, null, 'TOTAL BY DEPT', sum(sal)
  from emp
 group by deptno

DEPTNO JOB       CATEGORY                  SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
       ANALYST   TOTAL BY JOB             6000
       CLERK     TOTAL BY JOB             4150
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
```

```
    10           TOTAL BY DEPT            8750
    20           TOTAL BY DEPT            10875
    30           TOTAL BY DEPT             9400
```

The final step is to use UNION ALL to append the sum of all salaries:

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
 union all
select deptno, null, 'TOTAL BY DEPT', sum(sal)
  from emp
 group by deptno
 union all
select null,null, 'GRAND TOTAL FOR TABLE', sum(sal)
  from emp

DEPTNO JOB       CATEGORY                  SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
       ANALYST   TOTAL BY JOB             6000
       CLERK     TOTAL BY JOB             4150
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
    10           TOTAL BY DEPT            8750
    20           TOTAL BY DEPT           10875
    30           TOTAL BY DEPT            9400
                 GRAND TOTAL FOR TABLE   29025
```

12.14 Identifying Rows That Are Not Subtotals

Problem

You've used the CUBE extension of the GROUP BY clause to create a report, and you need a way to differentiate between rows that would be generated by a normal

GROUP BY clause and those rows that have been generated as a result of using CUBE or ROLLUP.

The following is the result set from a query using the CUBE extension to GROUP BY to create a breakdown of the salaries in table EMP:

```
DEPTNO JOB           SAL
------ --------- -------
                   29025
       CLERK        4150
       ANALYST      6000
       MANAGER      8275
       SALESMAN     5600
       PRESIDENT    5000
    10              8750
    10 CLERK        1300
    10 MANAGER      2450
    10 PRESIDENT    5000
    20             10875
    20 CLERK        1900
    20 ANALYST      6000
    20 MANAGER      2975
    30              9400
    30 CLERK         950
    30 MANAGER      2850
    30 SALESMAN     5600
```

This report includes the sum of all salaries by DEPTNO and JOB (for each JOB per DEPTNO), the sum of all salaries by DEPTNO, the sum of all salaries by JOB, and finally a grand total (the sum of all salaries in table EMP). You want to clearly identify the different levels of aggregation. You want to be able to identify which category an aggregated value belongs to (i.e., does a given value in the SAL column represent a total by DEPTNO? By JOB? The grand total?). You would like to return the following result set:

```
DEPTNO JOB           SAL DEPTNO_SUBTOTALS JOB_SUBTOTALS
------ --------- ------- ---------------- -------------
                   29025                1             1
       CLERK        4150                1             0
       ANALYST      6000                1             0
       MANAGER      8275                1             0
       SALESMAN     5600                1             0
       PRESIDENT    5000                1             0
    10              8750                0             1
    10 CLERK        1300                0             0
    10 MANAGER      2450                0             0
    10 PRESIDENT    5000                0             0
    20             10875                0             1
    20 CLERK        1900                0             0
    20 ANALYST      6000                0             0
    20 MANAGER      2975                0             0
```

```
30              9400                0             1
30 CLERK         950                0             0
30 MANAGER      2850                0             0
30 SALESMAN     5600                0             0
```

Solution

Use the GROUPING function to identify which values exist due to CUBE's or ROLLUP's creation of subtotals, or *superaggregate* values. The following is an example for PostgreSQL, DB2, and Oracle:

```
1 select deptno, jo) sal,
2        grouping(deptno) deptno_subtotals,
3        grouping(job) job_subtotals
4   from emp
5  group by cube(deptno,job)
```

The only difference between the SQL Server solution and that for DB2 and Oracle lies in how the CUBE/ROLLUP clauses are written:

```
1 select deptno, job, sum(sal) sal,
2        grouping(deptno) deptno_subtotals,
3        grouping(job) job_subtotals
4   from emp
5  group by deptno,job with cube
```

This recipe is meant to highlight the use of CUBE and GROUPING when working with subtotals. As of the time of this writing, MySQL doesn't support either CUBE or GROUPING.

Discussion

If DEPTNO_SUBTOTALS is 0 and JOB_SUBTOTALS is 1 (in which case JOB is NULL), the value of SAL represents a subtotal of salaries by DEPTNO created by CUBE. If JOB_SUBTOTALS is 0 and DEPTNO_SUBTOTALS is 1 (in which case DEPTNO is NULL), the value of SAL represents a subtotal of salaries by JOB created by CUBE. Rows with 0 for both DEPTNO_SUBTOTALS and JOB_SUBTOTALS represent rows created by regular aggregation (the sum of SAL for each DEPTNO/JOB combination).

12.15 Using Case Expressions to Flag Rows

Problem

You want to map the values in a column, perhaps the EMP table's JOB column, into a series of "Boolean" flags. For example, you want to return the following result set:

```
ENAME  IS_CLERK IS_SALES IS_MGR IS_ANALYST IS_PREZ
------ -------- -------- ------ ---------- -------
KING          0        0      0          0       1
SCOTT         0        0      0          1       0
FORD          0        0      0          1       0
JONES         0        0      1          0       0
BLAKE         0        0      1          0       0
CLARK         0        0      1          0       0
ALLEN         0        1      0          0       0
WARD          0        1      0          0       0
MARTIN        0        1      0          0       0
TURNER        0        1      0          0       0
SMITH         1        0      0          0       0
MILLER        1        0      0          0       0
ADAMS         1        0      0          0       0
JAMES         1        0      0          0       0
```

Such a result set can be useful for debugging and to provide yourself a view of the data different from what you'd see in a more typical result set.

Solution

Use a CASE expression to evaluate each employee's JOB, and return a 1 or 0 to signify their JOB. You'll need to write one CASE expression, and thus create one column for each possible job:

```
 1 select ename,
 2        case when job = 'CLERK'
 3             then 1 else 0
 4        end as is_clerk,
 5        case when job = 'SALESMAN'
 6             then 1 else 0
 7        end as is_sales,
 8        case when job = 'MANAGER'
 9             then 1 else 0
10        end as is_mgr,
11        case when job = 'ANALYST'
12             then 1 else 0
13        end as is_analyst,
14        case when job = 'PRESIDENT'
15             then 1 else 0
16        end as is_prez
17   from emp
18  order by 2,3,4,5,6
```

Discussion

The solution code is pretty much self-explanatory. If you are having trouble understanding it, simply add JOB to the SELECT clause:

```
select ename,
       job,
       case when job = 'CLERK'
            then 1 else 0
       end as is_clerk,
       case when job = 'SALESMAN'
            then 1 else 0
       end as is_sales,
       case when job = 'MANAGER'
            then 1 else 0
       end as is_mgr,
       case when job = 'ANALYST'
           then 1 else 0
       end as is_analyst,
       case when job = 'PRESIDENT'
           then 1 else 0
       end as is_prez
  from emp
 order by 2

ENAME  JOB       IS_CLERK IS_SALES IS_MGR IS_ANALYST IS_PREZ
------ --------- -------- -------- ------ ---------- -------
SCOTT  ANALYST          0        0      0          1       0
FORD   ANALYST          0        0      0          1       0
SMITH  CLERK            1        0      0          0       0
ADAMS  CLERK            1        0      0          0       0
MILLER CLERK            1        0      0          0       0
JAMES  CLERK            1        0      0          0       0
JONES  MANAGER          0        0      1          0       0
CLARK  MANAGER          0        0      1          0       0
BLAKE  MANAGER          0        0      1          0       0
KING   PRESIDENT        0        0      0          0       1
ALLEN  SALESMAN         0        1      0          0       0
MARTIN SALESMAN         0        1      0          0       0
TURNER SALESMAN         0        1      0          0       0
WARD   SALESMAN         0        1      0          0       0
```

12.16 Creating a Sparse Matrix

Problem

You want to create a sparse matrix, such as the following one transposing the DEPTNO and JOB columns of table EMP:

```
D10        D20        D30        CLERKS MGRS  PREZ ANALS SALES
---------- ---------- ---------- ------ ----- ---- ----- ------
           SMITH                 SMITH
                      ALLEN                              ALLEN
                      WARD                               WARD
           JONES                        JONES
                      MARTIN                             MARTIN
                      BLAKE             BLAKE
CLARK                                   CLARK
           SCOTT                                  SCOTT
KING                                         KING
                      TURNER                             TURNER
           ADAMS                 ADAMS
                      JAMES      JAMES
           FORD                                   FORD
MILLER                           MILLER
```

Solution

Use CASE expressions to create a sparse row-to-column transformation:

```
1 select case deptno when 10 then ename end as d10,
2        case deptno when 20 then ename end as d20,
3        case deptno when 30 then ename end as d30,
4        case job when 'CLERK' then ename end as clerks,
5        case job when 'MANAGER' then ename end as mgrs,
6        case job when 'PRESIDENT' then ename end as prez,
7        case job when 'ANALYST' then ename end as anals,
8        case job when 'SALESMAN' then ename end as sales
9   from emp
```

Discussion

To transform the DEPTNO and JOB rows to columns, simply use a CASE expression to evaluate the possible values returned by those rows. That's all there is to it. As an aside, if you want to "densify" the report and get rid of some of those NULL rows, you would need to find something to group by. For example, use the window function ROW_NUMBER OVER to assign a ranking for each employee per DEPTNO, and then use the aggregate function MAX to rub out some of the NULLs:

```
select max(case deptno when 10 then ename end) d10,
       max(case deptno when 20 then ename end) d20,
       max(case deptno when 30 then ename end) d30,
       max(case job when 'CLERK' then ename end) clerks,
       max(case job when 'MANAGER' then ename end) mgrs,
       max(case job when 'PRESIDENT' then ename end) prez,
       max(case job when 'ANALYST' then ename end) anals,
       max(case job when 'SALESMAN' then ename end) sales
  from (
select deptno, job, ename,
       row_number()over(partition by deptno order by empno) rn
```

```
  from emp
       ) x
 group by rn

D10        D20        D30        CLERKS MGRS  PREZ ANALS SALES
---------- ---------- ---------- ------ ----- ---- ----- ------
CLARK      SMITH      ALLEN      SMITH  CLARK            ALLEN
KING       JONES      WARD              JONES KING       WARD
MILLER     SCOTT      MARTIN     MILLER       SCOTT      MARTIN
           ADAMS      BLAKE      ADAMS  BLAKE
           FORD       TURNER                 FORD       TURNER
                      JAMES      JAMES
```

12.17 Grouping Rows by Units of Time

Problem

You want to summarize data by some interval of time. For example, you have a transaction log and want to summarize transactions by five-second intervals. The rows in table TRX_LOG are shown here:

```
select trx_id,
       trx_date,
       trx_cnt
  from trx_log
TRX_ID TRX_DATE                TRX_CNT
------ -------------------- ----------
     1 28-JUL-2020 19:03:07         44
     2 28-JUL-2020 19:03:08         18
     3 28-JUL-2020 19:03:09         23
     4 28-JUL-2020 19:03:10         29
     5 28-JUL-2020 19:03:11         27
     6 28-JUL-2020 19:03:12         45
     7 28-JUL-2020 19:03:13         45
     8 28-JUL-2020 19:03:14         32
     9 28-JUL-2020 19:03:15         41
    10 28-JUL-2020 19:03:16         15
    11 28-JUL-2020 19:03:17         24
    12 28-JUL-2020 19:03:18         47
    13 28-JUL-2020 19:03:19         37
    14 28-JUL-2020 19:03:20         48
    15 28-JUL-2020 19:03:21         46
    16 28-JUL-2020 19:03:22         44
    17 28-JUL-2020 19:03:23         36
    18 28-JUL-2020 19:03:24         41
    19 28-JUL-2020 19:03:25         33
    20 28-JUL-2020 19:03:26         19
```

You want to return the following result set:

```
GRP TRX_START            TRX_END                   TOTAL
--- -------------------- -------------------- ----------
  1 28-JUL-2020 19:03:07 28-JUL-2020 19:03:11        141
  2 28-JUL-2020 19:03:12 28-JUL-2020 19:03:16        178
  3 28-JUL-2020 19:03:17 28-JUL-2020 19:03:21        202
  4 28-JUL-2020 19:03:22 28-JUL-2020 19:03:26        173
```

Solution

Group the entries into five row buckets. There are several ways to accomplish that logical grouping; this recipe does so by dividing the TRX_ID values by five, using a technique shown earlier in Recipe 12.7.

Once you've created the "groups," use the aggregate functions MIN, MAX, and SUM to find the start time, end time, and total number of transactions for each "group" (SQL Server users should use CEILING instead of CEIL):

```
select ceil(trx_id/5.0) as grp,
       min(trx_date)    as trx_start,
       max(trx_date)    as trx_end,
       sum(trx_cnt)     as total
  from trx_log
group by ceil(trx_id/5.0)
```

Discussion

The first step, and the key to the whole solution, is to logically group the rows together. By dividing by five and taking the smallest whole number greater than the quotient, you can create logical groups. For example:

```
select trx_id,
       trx_date,
       trx_cnt,
       trx_id/5.0 as val,
       ceil(trx_id/5.0) as grp
  from trx_log
TRX_ID TRX_DATE             TRX_CNT     VAL GRP
------ -------------------- -------  ------ ---
     1 28-JUL-2020 19:03:07      44     .20   1
     2 28-JUL-2020 19:03:08      18     .40   1
     3 28-JUL-2020 19:03:09      23     .60   1
     4 28-JUL-2020 19:03:10      29     .80   1
     5 28-JUL-2020 19:03:11      27    1.00   1
     6 28-JUL-2020 19:03:12      45    1.20   2
     7 28-JUL-2020 19:03:13      45    1.40   2
     8 28-JUL-2020 19:03:14      32    1.60   2
     9 28-JUL-2020 19:03:15      41    1.80   2
    10 28-JUL-2020 19:03:16      15    2.00   2
    11 28-JUL-2020 19:03:17      24    2.20   3
    12 28-JUL-2020 19:03:18      47    2.40   3
    13 28-JUL-2020 19:03:19      37    2.60   3
```

```
    14 28-JUL-2020 19:03:20      48    2.80   3
    15 28-JUL-2020 19:03:21      46    3.00   3
    16 28-JUL-2020 19:03:22      44    3.20   4
    17 28-JUL-2020 19:03:23      36    3.40   4
    18 28-JUL-2020 19:03:24      41    3.60   4
    19 28-JUL-2020 19:03:25      33    3.80   4
    20 28-JUL-2020 19:03:26      19    4.00   4
```

The last step is to apply the appropriate aggregate functions to find the total number of transactions per five seconds, along with the start and end times for each transaction:

```
select ceil(trx_id/5.0) as grp,
       min(trx_date) as trx_start,
       max(trx_date) as trx_end,
       sum(trx_cnt) as total
  from trx_log
 group by ceil(trx_id/5.0)
GRP TRX_START            TRX_END                   TOTAL
--- -------------------- -------------------- ----------
  1 28-JUL-2020 19:03:07 28-JUL-2005 19:03:11        141
  2 28-JUL-2020 19:03:12 28-JUL-2005 19:03:16        178
  3 28-JUL-2020 19:03:17 28-JUL-2005 19:03:21        202
  4 28-JUL-2020 19:03:22 28-JUL-2005 19:03:26        173
```

If your data is slightly different (perhaps you don't have an ID for each row), you can always "group" by dividing the seconds of each TRX_DATE row by five to create a similar grouping. Then you can include the hour for each TRX_DATE and group by the actual hour and logical "grouping," GRP. The following is an example of this technique (using Oracle's TO_CHAR and TO_NUMBER functions, you would use the appropriate date and character formatting functions for your platform):

```
select trx_date,trx_cnt,
       to_number(to_char(trx_date,'hh24')) hr,
       ceil(to_number(to_char(trx_date-1/24/60/60,'miss'))/5.0) grp
  from trx_log

TRX_DATE             20   TRX_CNT         HR         GRP
-------------------- ---------- ----------  ----------
28-JUL-2020 19:03:07         44         19          62
28-JUL-2020 19:03:08         18         19          62
28-JUL-2020 19:03:09         23         19          62
28-JUL-2020 19:03:10         29         19          62
28-JUL-2020 19:03:11         27         19          62
28-JUL-2020 19:03:12         45         19          63
28-JUL-2020 19:03:13         45         19          63
28-JUL-2020 19:03:14         32         19          63
28-JUL-2020 19:03:15         41         19          63
28-JUL-2020 19:03:16         15         19          63
28-JUL-2020 19:03:17         24         19          64
28-JUL-2020 19:03:18         47         19          64
28-JUL-2020 19:03:19         37         19          64
```

```
28-JUL-2020 19:03:20         48         19          64
28-JUL-2020 19:03:21         46         19          64
28-JUL-2020 19:03:22         44         19          65
28-JUL-2020 19:03:23         36         19          65
28-JUL-2020 19:03:24         41         19          65
28-JUL-2020 19:03:25         33         19          65
28-JUL-2020 19:03:26         19         19          65
```

Regardless of the actual values for GRP, the key here is that you are grouping for every five seconds. From there you can apply the aggregate functions in the same way as in the original solution:

```
select hr,grp,sum(trx_cnt) total
  from (
select trx_date,trx_cnt,
       to_number(to_char(trx_date,'hh24')) hr,
       ceil(to_number(to_char(trx_date-1/24/60/60,'miss'))/5.0) grp
  from trx_log
       ) x
 group by hr,grp
HR        GRP       TOTAL
-- ----------  ----------
19         62         141
19         63         178
19         64         202
19         65         173
```

Including the hour in the grouping is useful if your transaction log spans hours. In DB2 and Oracle, you can also use the window function SUM OVER to produce the same result. The following query returns all rows from TRX_LOG along with a running total for TRX_CNT by logical "group," and the TOTAL for TRX_CNT for each row in the "group":

```
select trx_id, trx_date, trx_cnt,
       sum(trx_cnt)over(partition by ceil(trx_id/5.0)
                        order by trx_date
                        range between unbounded preceding
                          and current row) runing_total,
       sum(trx_cnt)over(partition by ceil(trx_id/5.0)) total,
       case when mod(trx_id,5.0) = 0 then 'X' end grp_end
  from trx_log

TRX_ID TRX_DATE                TRX_CNT RUNING_TOTAL      TOTAL GRP_END
------ -------------------- ---------- ------------ ---------- -------
     1 28-JUL-2020 19:03:07         44           44        141
     2 28-JUL-2020 19:03:08         18           62        141
     3 28-JUL-2020 19:03:09         23           85        141
     4 28-JUL-2020 19:03:10         29          114        141
     5 28-JUL-2020 19:03:11         27          141        141  X
     6 28-JUL-2020 19:03:12         45           45        178
     7 28-JUL-2020 19:03:13         45           90        178
     8 28-JUL-2020 19:03:14         32          122        178
```

```
     9 28-JUL-2020 19:03:15         41          163       178
    10 28-JUL-2020 19:03:16         15          178       178  X
    11 28-JUL-2020 19:03:17         24           24       202
    12 28-JUL-2020 19:03:18         47           71       202
    13 28-JUL-2020 19:03:19         37          108       202
    14 28-JUL-2020 19:03:20         48          156       202
    15 28-JUL-2020 19:03:21         46          202       202  X
    16 28-JUL-2020 19:03:22         44           44       173
    17 28-JUL-2020 19:03:23         36           80       173
    18 28-JUL-2020 19:03:24         41          121       173
    19 28-JUL-2020 19:03:25         33          154       173
    20 28-JUL-2020 19:03:26         19          173       173  X
```

12.18 Performing Aggregations over Different Groups/ Partitions Simultaneously

Problem

You want to aggregate over different dimensions at the same time. For example, you want to return a result set that lists each employee's name, their department, the number of employees in their department (themselves included), the number of employees that have the same job (themselves included in this count as well), and the total number of employees in the EMP table. The result set should look like the following:

```
ENAME  DEPTNO DEPTNO_CNT JOB       JOB_CNT   TOTAL
------ ------ ---------- --------- -------- ------
MILLER     10          3 CLERK            4     14
CLARK      10          3 MANAGER          3     14
KING       10          3 PRESIDENT        1     14
SCOTT      20          5 ANALYST          2     14
FORD       20          5 ANALYST          2     14
SMITH      20          5 CLERK            4     14
JONES      20          5 MANAGER          3     14
ADAMS      20          5 CLERK            4     14
JAMES      30          6 CLERK            4     14
MARTIN     30          6 SALESMAN         4     14
TURNER     30          6 SALESMAN         4     14
WARD       30          6 SALESMAN         4     14
ALLEN      30          6 SALESMAN         4     14
BLAKE      30          6 MANAGER          3     14
```

Solution

Use the COUNT OVER window function while specifying different partitions, or groups of data, on which to perform aggregation:

```
select ename,
       deptno,
       count(*)over(partition by deptno) deptno_cnt,
```

```
        job,
        count(*)over(partition by job) job_cnt,
        count(*)over() total
    from emp
```

Discussion

This example really shows off the power and convenience of window functions. By simply specifying different partitions or groups of data to aggregate, you can create immensely detailed reports without having to self-join over and over, and without having to write cumbersome and perhaps poorly performing subqueries in your SELECT list. All the work is done by the window function COUNT OVER. To understand the output, focus on the OVER clause for a moment for each COUNT operation:

```
count(*)over(partition by deptno)

count(*)over(partition by job)

count(*)over()
```

Remember the main parts of the OVER clause: the PARTITION BY subclause, dividing the query into partitions; and the ORDER BY subclause, defining the logical order. Look at the first COUNT, which partitions by DEPTNO. The rows in table EMP will be grouped by DEPTNO, and the COUNT operation will be performed on all the rows in each group. Since there is no frame or window clause specified (no ORDER BY), all the rows in the group are counted. The PARTITION BY clause finds all the unique DEPTNO values, and then the COUNT function counts the number of rows having each value. In the specific example of COUNT(*)OVER(PARTITION BY DEPTNO), the PARTITION BY clause identifies the partitions or groups to be values 10, 20, and 30.

The same processing is applied to the second COUNT, which partitions by JOB. The last count does not partition by anything and simply has an empty parentheses. An empty parentheses implies "the whole table." So, whereas the two prior COUNTs aggregate values based on the defined groups or partitions, the final COUNT counts all rows in table EMP.

Keep in mind that window functions are applied after the WHERE clause. If you were to filter the result set in some way, for example, excluding all employees in DEPTNO 10, the value for TOTAL would not be 14—it would be 11. To filter results after window functions have been evaluated, you must make your windowing query into an inline view and then filter on the results from that view.

12.19 Performing Aggregations over a Moving Range of Values

Problem

You want to compute a moving aggregation, such as a moving sum on the salaries in table EMP. You want to compute a sum for every 90 days, starting with the HIREDATE of the first employee. You want to see how spending has fluctuated for every 90-day period between the first and last employee hired. You want to return the following result set:

```
HIREDATE        SAL SPENDING_PATTERN
----------- ------- ----------------
17-DEC-200     800              800
20-FEB-2011    1600             2400
22-FEB-2011    1250             3650
02-APR-2011    2975             5825
01-MAY-2011    2850             8675
09-JUN-2011    2450             8275
08-SEP-2011    1500             1500
28-SEP-2011    1250             2750
17-NOV-2011    5000             7750
03-DEC-2011     950            11700
03-DEC-2011    3000            11700
23-JAN-2012    1300            10250
09-DEC-2012    3000             3000
12-JAN-2013    1100             4100
```

Solution

Being able to specify a moving window in the framing or windowing clause of window functions makes this problem easy to solve, if your RDBMS supports such functions. The key is to order by HIREDATE in your window function and then specify a window of 90 days starting from the earliest employee hired. The sum will be computed using the salaries of employees hired up to 90 days prior to the current employee's HIREDATE (the current employee is included in the sum). If you do not have window functions available, you can use scalar subqueries, but the solution will be more complex.

DB2 and Oracle

For DB2 and Oracle, use the window function SUM OVER and order by HIREDATE. Specify a range of 90 days in the window or "framing" clause to allow the sum to be computed for each employee's salary and to include the salaries of all employees hired up to 90 days earlier. Because DB2 does not allow you to specify HIREDATE in the

ORDER BY clause of a window function (line 3 in the following code), you can order by DAYS(HIREDATE) instead:

```
1 select hiredate,
2        sal,
3        sum(sal)over(order by days(hiredate)
4                       range between 90 preceding
5                         and current row) spending_pattern
6   from emp e
```

The Oracle solution is more straightforward than DB2's, because Oracle allows window functions to order by datetime types:

```
1 select hiredate,
2        sal,
3        sum(sal)over(order by hiredate
4                       range between 90 preceding
5                         and current row) spending_pattern
6   from emp e
```

MySQL

Use the window function with slightly altered syntax:

```
1  select hiredate,
2          sal,
3          sum(sal)over(order by hiredate
4              range interval 90 day preceding ) spending_pattern
5  from emp e
```

PostgreSQL and SQL Server

Use a scalar subquery to sum the salaries of all employees hired up to 90 days prior to the day each employee was hired:

```
1 select e.hiredate,
2        e.sal,
3        (select sum(sal) from emp d
4          whered.hiredate between e.hiredate-90
5                              and e.hiredate) as spending_pattern
6   from emp e
7  order by 1
```

Discussion

DB2, MySQL, and Oracle

DB2, MySQL, and Oracle share the same logical solution. The only minor differences between the solutions are in how you specify HIREDATE in the ORDER BY clause of the window function and the syntax of specifying the time interval in MySQL. At the time of this book's writing, DB2 doesn't allow a DATE value in such an ORDER BY

clause if you are using a numeric value to set the window's range. (For example, RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW allows you to order by a date, but RANGE BETWEEN 90 PRECEDING AND CURRENT ROW does not.)

To understand what the solution query is doing, you simply need to understand what the window clause is doing. The window you are defining orders the salaries for all employees by HIREDATE. Then the function computes a sum. The sum is not computed for all salaries. Instead, the processing is as follows:

1. The salary of the first employee hired is evaluated. Since no employees were hired before the first employee, the sum at this point is simply the first employee's salary.
2. The salary of the next employee (by HIREDATE) is evaluated. This employee's salary is included in the moving sum along with any other employees who were hired up to 90 days prior.

The HIREDATE of the first employee is December 17, 2010, and the HIREDATE of the next hired employee is February 20, 2011. The second employee was hired less than 90 days after the first employee, and thus the moving sum for the second employee is 2400 (1600 + 800). If you are having trouble understanding where the values in SPENDING_PATTERN come from, examine the following query and result set:

```
select distinct
       dense_rank()over(order by e.hiredate) window,
       e.hiredate current_hiredate,
       d.hiredate hiredate_within_90_days,
       d.sal sals_used_for_sum
  from emp e,
       emp d
where d.hiredate between e.hiredate-90 and e.hiredate

WINDOW CURRENT_HIREDATE HIREDATE_WITHIN_90_DAYS SALS_USED_FOR_SUM
------ --------------- ----------------------- -----------------
     1 17-DEC-2010      17-DEC-2010                           800
     2 20-FEB-2011      17-DEC-2010                           800
     2 20-FEB-2011      20-FEB-2011                          1600
     3 22-FEB-2011      17-DEC-2010                           800
     3 22-FEB-2011      20-FEB-2011                          1600
     3 22-FEB-2011      22-FEB-2011                          1250
     4 02-APR-2011      20-FEB-2011                          1600
     4 02-APR-2011      22-FEB-2011                          1250
     4 02-APR-2011      02-APR-2011                          2975
     5 01-MAY-2011      20-FEB-2011                          1600
     5 01-MAY-2011      22-FEB-2011                          1250
     5 01-MAY-2011      02-APR-2011                          2975
     5 01-MAY-2011      01-MAY-2011                          2850
```

```
     6 09-JUN-2011      02-APR-2011                          2975
     6 09-JUN-2011      01-MAY-2011                          2850
     6 09-JUN-2011      09-JUN-2011                          2450
     7 08-SEP-2011      08-SEP-2011                          1500
     8 28-SEP-2011      08-SEP-2011                          1500
     8 28-SEP-2011      28-SEP-2011                          1250
     9 17-NOV-2011      08-SEP-2011                          1500
     9 17-NOV-2011      28-SEP-2011                          1250
     9 17-NOV-2011      17-NOV-2011                          5000
    10 03-DEC-2011      08-SEP-2011                          1500
    10 03-DEC-2011      28-SEP-2011                          1250
    10 03-DEC-2011      17-NOV-2011                          5000
    10 03-DEC-2011      03-DEC-2011                           950
    10 03-DEC-2011      03-DEC-2011                          3000
    11 23-JAN-2012      17-NOV-2011                          5000
    11 23-JAN-2012      03-DEC-2011                           950
    11 23-JAN-2012      03-DEC-2011                          3000
    11 23-JAN-2012      23-JAN-2012                          1300
    12 09-DEC-2012      09-DEC-2012                          3000
    13 12-JAN-2013      09-DEC-2012                          3000
    13 12-JAN-2013      12-JAN-2013                          1100
```

If you look at the WINDOW column, only those rows with the same WINDOW value will be considered for each sum. Take, for example, WINDOW 3. The salaries used for the sum for that window are 800, 1600, and 1250, which total 3650. If you look at the final result set in the "Problem" section, you'll see the SPENDING_PATTERN for February 22, 2011 (WINDOW 3) is 3650. As proof, to verify that the previous self-join includes the correct salaries for the windows defined, simply sum the values in SALS_USED_FOR_SUM and group by CURRENT_DATE. The result should be the same as the result set shown in the "Problem" section (with the duplicate row for December 3, 2011, filtered out):

```
select current_hiredate,
       sum(sals_used_for_sum) spending_pattern
  from (
select distinct
       dense_rank()over(order by e.hiredate) window,
       e.hiredate current_hiredate,
       d.hiredate hiredate_within_90_days,
       d.sal sals_used_for_sum
  from emp e,
       emp d
  where d.hiredate between e.hiredate-90 and e.hiredate
        ) x
  group by current_hiredate

CURRENT_HIREDATE SPENDING_PATTERN
---------------- ----------------
17-DEC-2010                  800
20-FEB-2011                 2400
22-FEB-2011                 3650
```

```
02-APR-2011                  5825
01-MAY-2011                  8675
09-JUN-2011                  8275
08-SEP-2011                  1500
28-SEP-2011                  2750
17-NOV-2011                  7750
03-DEC-2011                 11700
23-JAN-2012                 10250
09-DEC-2012                  3000
12-JAN-2013                  4100
```

PostgreSQL and SQL Server

The key to this solution is to use a scalar subquery (a self-join will work as well) while using the aggregate function SUM to compute a sum for every 90 days based on HIREDATE. If you are having trouble seeing how this works, simply convert the solution to a self-join and examine which rows are included in the computations. Consider the following result set, which returns the same result set as that in the solution:

```
select e.hiredate,
       e.sal,
       sum(d.sal) as spending_pattern
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 group by e.hiredate,e.sal
 order by 1\

HIREDATE      SAL   SPENDING_PATTERN
----------- -----   ----------------
17-DEC-2010   800                800
20-FEB-2011  1600               2400
22-FEB-2011  1250               3650
02-APR-2011  2975               5825
01-MAY-2011  2850               8675
09-JUN-2011  2450               8275
08-SEP-2011  1500               1500
28-SEP-2011  1250               2750
17-NOV-2011  5000               7750
03-DEC-2011   950              11700
03-DEC-2011  3000              11700
23-JAN-2012  1300              10250
09-DEC-2012  3000               3000
12-JAN-2013  1100               4100
```

If it is still unclear, simply remove the aggregation and start with the Cartesian product. The first step is to generate a Cartesian product using table EMP so that each HIREDATE can be compared with all the other HIREDATEs. (Only a snippet of the result set is shown here because there are 196 rows (14 × 14) returned by a Cartesian of EMP):

```
select e.hiredate,
       e.sal,
       d.sal,
       d.hiredate
  from emp e, emp d

HIREDATE      SAL      SAL HIREDATE
----------- -----    ----- -----------
17-DEC-2010   800      800 17-DEC-2010
17-DEC-2010   800     1600 20-FEB-2011
17-DEC-2010   800     1250 22-FEB-2011
17-DEC-2010   800     2975 02-APR-2011
17-DEC-2010   800     1250 28-SEP-2011
17-DEC-2010   800     2850 01-MAY-2011
17-DEC-2010   800     2450 09-JUN-2011
17-DEC-2010   800     3000 09-DEC-2012
17-DEC-2010   800     5000 17-NOV-2011
17-DEC-2010   800     1500 08-SEP-2011
17-DEC-2010   800     1100 12-JAN-2013
17-DEC-2010   800      950 03-DEC-2011
17-DEC-2010   800     3000 03-DEC-2011
17-DEC-2010   800     1300 23-JAN-2012
20-FEB-2011  1600      800 17-DEC-2010
20-FEB-2011  1600     1600 20-FEB-2011
20-FEB-2011  1600     1250 22-FEB-2011
20-FEB-2011  1600     2975 02-APR-2011
20-FEB-2011  1600     1250 28-SEP-2011
20-FEB-2011  1600     2850 01-MAY-2011
20-FEB-2011  1600     2450 09-JUN-2011
20-FEB-2011  1600     3000 09-DEC-2012
20-FEB-2011  1600     5000 17-NOV-2011
20-FEB-2011  1600     1500 08-SEP-2011
20-FEB-2011  1600     1100 12-JAN-2013
20-FEB-2011  1600      950 03-DEC-2011
20-FEB-2011  1600     3000 03-DEC-2011
20-FEB-2011  1600     1300 23-JAN-2012
```

If you examine the previous result set, you'll notice that there is no HIREDATE 90 days earlier or equal to December 17, except for December 17. So, the sum for that row should be only 800. If you examine the next HIREDATE, February 20, you'll notice that there is one HIREDATE that falls within the 90-day window (within 90 days prior), and that is December 17. If you sum the SAL from December 17 with the SAL from February 20 (because we are looking for HIREDATEs equal to each HIREDATE or within 90 days earlier), you get 2400, which happens to be the final result for that HIREDATE.

Now that you know how it works, use a filter in the WHERE clause to return for each HIREDATE and HIREDATE that is equal to it or is no more than 90 days earlier:

```
select e.hiredate,
       e.sal,
       d.sal sal_to_sum,
       d.hiredate within_90_days
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 order by 1
HIREDATE      SAL SAL_TO_SUM WITHIN_90_DAYS
----------- ----- ---------- --------------
17-DEC-2010   800        800    17-DEC-2010
20-FEB-2011  1600        800    17-DEC-2010
20-FEB-2011  1600       1600    20-FEB-2011
22-FEB-2011  1250        800    17-DEC-2010
22-FEB-2011  1250       1600    20-FEB-2011
22-FEB-2011  1250       1250    22-FEB-2011
02-APR-2011  2975       1600    20-FEB-2011
02-APR-2011  2975       1250    22-FEB-2011
02-APR-2011  2975       2975    02-APR-2011
01-MAY-2011  2850       1600    20-FEB-2011
01-MAY-2011  2850       1250    22-FEB-2011
01-MAY-2011  2850       2975    02-APR-2011
01-MAY-2011  2850       2850    01-MAY-2011
09-JUN-2011  2450       2975    02-APR-2011
09-JUN-2011  2450       2850    01-MAY-2011
09-JUN-2011  2450       2450    09-JUN-2011
08-SEP-2011  1500       1500    08-SEP-2011
28-SEP-2011  1250       1500    08-SEP-2011
28-SEP-2011  1250       1250    28-SEP-2011
17-NOV-2011  5000       1500    08-SEP-2011
17-NOV-2011  5000       1250    28-SEP-2011
17-NOV-2011  5000       5000    17-NOV-2011
03-DEC-2011   950       1500    08-SEP-2011
03-DEC-2011   950       1250    28-SEP-2011
03-DEC-2011   950       5000    17-NOV-2011
03-DEC-2011   950        950    03-DEC-2011
03-DEC-2011   950       3000    03-DEC-2011
03-DEC-2011  3000       1500    08-SEP-2011
03-DEC-2011  3000       1250    28-SEP-2011
03-DEC-2011  3000       5000    17-NOV-2011
03-DEC-2011  3000        950    03-DEC-2011
03-DEC-2011  3000       3000    03-DEC-2011
23-JAN-2012  1300       5000    17-NOV-2011
23-JAN-2012  1300        950    03-DEC-2011
23-JAN-2012  1300       3000    03-DEC-2011
23-JAN-2012  1300       1300    23-JAN-2012
09-DEC-2012  3000       3000    09-DEC-2012
12-JAN-2013  1100       3000    09-DEC-2012
12-JAN-2013  1100       1100    12-JAN-2013
```

Now that you know which SALs are to be included in the moving window of summation, simply use the aggregate function SUM to produce a more expressive result set:

```
select e.hiredate,
       e.sal,
       sum(d.sal) as spending_pattern
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 group by e.hiredate,e.sal
 order by 1
```

If you compare the result set for the previous query and the result set for the query shown here (which is the original solution presented), you will see they are the same:

```
select e.hiredate,
       e.sal,
       (select sum(sal) from emp d
        where d.hiredate between e.hiredate-90
                             and e.hiredate) as spending_pattern
  from emp e
 order by 1

HIREDATE      SAL SPENDING_PATTERN
----------- ----- ----------------
17-DEC-2010  800               800
20-FEB-2011 1600              2400
22-FEB-2011 1250              3650
02-APR-2011 2975              5825
01-MAY-2011 2850              8675
09-JUN-2011 2450              8275
08-SEP-2011 1500              1500
28-SEP-2011 1250              2750
17-NOV-2011 5000              7750
03-DEC-2011  950             11700
03-DEC-2011 3000             11700
23-JAN-2012 1300             10250
09-DEC-2012 3000              3000
12-JAN-2013 1100              4100
```

12.20 Pivoting a Result Set with Subtotals

Problem

You want to create a report containing subtotals and then transpose the results to provide a more readable report. For example, you've been asked to create a report that displays for each department, the managers in the department, and a sum of the salaries of the employees who work for those managers. Additionally, you want to return two subtotals: the sum of all salaries in each department for those employees who have managers, and a sum of all salaries in the result set (the sum of the department subtotals). You currently have the following report:

```
DEPTNO        MGR        SAL
------ ---------- ----------
    10       7782       1300
    10       7839       2450
    10                  3750
    20       7566       6000
    20       7788       1100
    20       7839       2975
    20       7902        800
    20                  10875
    30       7698       6550
    30       7839       2850
    30                  9400
                       24025
```

You want to provide a more readable report and want to transform the previous result set to the following, which makes the meaning of the report much clearer:

```
MGR     DEPT10     DEPT20     DEPT30      TOTAL
---- ---------- ---------- ---------- ----------
7566          0       6000          0
7698          0          0       6550
7782       1300          0          0
7788          0       1100          0
7839       2450       2975       2850
7902          0        800          0
           3750      10875       9400     24025
```

Solution

The first step is to generate subtotals using the ROLLUP extension to GROUP BY. The next step is to perform a classic pivot (aggregate and CASE expression) to create the desired columns for your report. The GROUPING function allows you to easily determine which values are subtotals (that is, exist because of ROLLUP and otherwise would not normally be there). Depending on how your RDBMS sorts NULL values, you may need to add an ORDER BY to the solution to allow it to look like the previous target result set.

DB2 and Oracle

Use the ROLLUP extension to GROUP BY and then use a CASE expression to format the data into a more readable report:

```
1 select mgr,
2        sum(case deptno when 10 then sal else 0 end) dept10,
3        sum(case deptno when 20 then sal else 0 end) dept20,
4        sum(case deptno when 30 then sal else 0 end) dept30,
5        sum(case flag when '11' then sal else null end) total
6   from (
7 select deptno,mgr,sum(sal) sal,
8        cast(grouping(deptno) as char(1))||
```

```
       cast(grouping(mgr) as char(1)) flag
  from emp
 where mgr is not null
 group by rollup(deptno,mgr)
       ) x
 group by mgr
```

SQL Server

Use the ROLLUP extension to GROUP BY and then use a CASE expression to format the data into a more readable report:

```
select mgr,
       sum(case deptno when 10 then sal else 0 end) dept10,
       sum(case deptno when 20 then sal else 0 end) dept20,
       sum(case deptno when 30 then sal else 0 end) dept30,
       sum(case flag when '11' then sal else null end) total
  from (
select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr) as char(1)) flag
  from emp
 where mgr is not null
 group by deptno,mgr with rollup
       ) x
 group by mgr
```

PostgreSQL

Use the ROLLUP extension to GROUP BY and then use a CASE expression to format the data into a more readable report:

```
  select mgr,
         sum(case deptno when 10 then sal else 0 end) dept10,
         sum(case deptno when 20 then sal else 0 end) dept20,
         sum(case deptno when 30 then sal else 0 end) dept30,
         sum(case flag when '11' then sal else null end) total
    from (
  select deptno,mgr,sum(sal) sal,
         concat(cast (grouping(deptno) as char(1)),
         cast(grouping(mgr) as char(1))) flag
  from emp
  where mgr is not null
  group by rollup (deptno,mgr)
       ) x
  group by mgr
```

MySQL

Use the ROLLUP extension to GROUP BY and then use a CASE expression to format the data into a more readable report:

```
1    select mgr,
2          sum(case deptno when 10 then sal else 0 end) dept10,
3          sum(case deptno when 20 then sal else 0 end) dept20,
4          sum(case deptno when 30 then sal else 0 end) dept30,
5          sum(case flag when '11' then sal else null end) total
6     from (
7    select  deptno,mgr,sum(sal) sal,
8            concat( cast(grouping(deptno) as char(1)) ,
9            cast(grouping(mgr) as char(1))) flag
10   from emp
11  where mgr is not null
12   group by deptno,mgr with rollup
13         ) x
14   group by mgr;
```

Discussion

The solutions provided here are identical except for the string concatenation and how GROUPING is specified. Because the solutions are so similar, the following discussion will refer to the SQL Server solution to highlight the intermediate result sets (the discussion is relevant to DB2 and Oracle as well).

The first step is to generate a result set that sums the SAL for the employees in each DEPTNO per MGR. The idea is to show how much the employees make under a particular manager in a particular department. For example, the following query will allow you to compare the salaries of employees who work for KING in DEPTNO 10 compared with those who work for KING in DEPTNO 30:

```
select deptno,mgr,sum(sal) sal
  from emp
 where mgr is not null
 group by mgr,deptno
 order by 1,2

DEPTNO        MGR        SAL
------ ---------- ----------
    10       7782       1300
    10       7839       2450
    20       7566       6000
    20       7788       1100
    20       7839       2975
    20       7902        800
    30       7698       6550
    30       7839       2850
```

The next step is to use the ROLLUP extension to GROUP BY to create subtotals for each DEPTNO and across all employees (who have a manager):

```
select deptno,mgr,sum(sal) sal
  from emp
 where mgr is not null
```

```
 group by deptno,mgr with rollup

DEPTNO        MGR        SAL
------ ---------- ----------
    10       7782       1300
    10       7839       2450
    10                  3750
    20       7566       6000
    20       7788       1100
    20       7839       2975
    20       7902        800
    20                 10875
    30       7698       6550
    30       7839       2850
    30                  9400
                       24025
```

With the subtotals created, you need a way to determine which values are in fact subtotals (created by ROLLUP) and which are results of the regular GROUP BY. Use the GROUPING function to create bitmaps to help identify the subtotal values from the regular aggregate values:

```
select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr) as char(1)) flag
  from emp
 where mgr is not null
 group by deptno,mgr with rollup

DEPTNO        MGR        SAL FLAG
------ ---------- ---------- ----
    10       7782       1300 00
    10       7839       2450 00
    10                  3750 01
    20       7566       6000 00
    20       7788       1100 00
    20       7839       2975 00
    20       7902        800 00
    20                 10875 01
    30       7698       6550 00
    30       7839       2850 00
    30                  9400 01
                       24025 11
```

If it isn't immediately obvious, the rows with a value of 00 for FLAG are the results of regular aggregation. The rows with a value of 01 for FLAG are the results of ROLLUP aggregating SAL by DEPTNO (since DEPTNO is listed first in the ROLLUP; if you switch the order, for example, GROUP BY MGR, DEPTNO WITH ROLLUP, you'd see quite different results). The row with a value of 11 for FLAG is the result of ROLLUP aggregating SAL over all rows.

At this point you have everything you need to create a beautified report by simply using CASE expressions. The goal is to provide a report that shows employee salaries for each manager across departments. If a manager does not have any subordinates in a particular department, a zero should be returned; otherwise, you want to return the sum of all salaries for that manager's subordinates in that department. Additionally, you want to add a final column, TOTAL, representing a sum of all the salaries in the report. The solution satisfying all these requirements is shown here:

```
select mgr,
       sum(case deptno when 10 then sal else 0 end) dept10,
       sum(case deptno when 20 then sal else 0 end) dept20,
       sum(case deptno when 30 then sal else 0 end) dept30,
       sum(case flag when '11' then sal else null end) total
  from (
select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr) as char(1)) flag
  from emp
 where mgr is not null
 group by deptno,mgr with rollup
       ) x
 group by mgr
 order by coalesce(mgr,9999)

MGR     DEPT10     DEPT20     DEPT30     TOTAL
---- ---------- ---------- ---------- ----------
7566          0       6000          0
7698          0          0       6550
7782       1300          0          0
7788          0       1100          0
7839       2450       2975       2850
7902          0        800          0
           3750      10875       9400    24025
```

12.21 Summing Up

Databases are for storing data, but eventually someone needs to retrieve the data and present it somewhere. The recipes in this chapter show a variety of important ways that data can be re-shaped or formatted to meet the needs of users. Apart from their general usefulness in giving users data in the form they need, these techniques play an important role in giving a database owner the ability to create a datawarehouse.

As you gain more experience in supporting users in the business, you will become more adept and extend the ideas here into more elaborate presentations.

CHAPTER 13

Hierarchical Queries

This chapter introduces recipes for expressing hierarchical relationships that you may have in your data. It is typical when working with hierarchical data to have more difficulty retrieving and displaying the data (as a hierarchy) than storing it.

Although it's only been a couple of years since MySQL added recursive CTEs, now that they are available it means that recursive CTEs are available in virtually every RDBMS. As a result, they are the gold standard for dealing with hierarchical queries, and this chapter will make liberal use of this capability to provide recipes to help you unravel the hierarchical structure of your data.

Before starting, examine table EMP and the hierarchical relationship between EMPNO and MGR:

```
select empno,mgr
  from emp
order by 2

     EMPNO        MGR
---------- ----------
      7788       7566
      7902       7566
      7499       7698
      7521       7698
      7900       7698
      7844       7698
      7654       7698
      7934       7782
      7876       7788
      7566       7839
      7782       7839
      7698       7839
      7369       7902
      7839
```

If you look carefully, you will see that each value for MGR is also an EMPNO, meaning the manager of each employee in table EMP is also an employee in table EMP and not stored somewhere else. The relationship between MGR and EMPNO is a parent-child relationship in that the value for MGR is the most immediate parent for a given EMPNO (it is also possible that the manager for a specific employee can have a manager as well, and those managers can in turn have managers, and so on, creating an *n*-tier hierarchy). If an employee has no manager, then MGR is NULL.

13.1 Expressing a Parent-Child Relationship

Problem

You want to include parent information along with data from child records. For example, you want to display each employee's name along with the name of their manager. You want to return the following result set:

```
EMPS_AND_MGRS
-----------------------------
FORD works for JONES
SCOTT works for JONES
JAMES works for BLAKE
TURNER works for BLAKE
MARTIN works for BLAKE
WARD works for BLAKE
ALLEN works for BLAKE
MILLER works for CLARK
ADAMS works for SCOTT
CLARK works for KING
BLAKE works for KING
JONES works for KING
SMITH works for FORD
```

Solution

Self-join EMP on MGR and EMPNO to find the name of each employee's manager. Then use your RDBMS's supplied function(s) for string concatenation to generate the strings in the desired result set.

DB2, Oracle, and PostgreSQL

Self-join on EMP. Then use the double vertical-bar (||) concatenation operator:

```
1 select a.ename || ' works for ' || b.ename as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

MySQL

Self-join on EMP. Then use the concatenation function CONCAT:

```
1 select concat(a.ename, ' works for ',b.ename) as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

SQL Server

Self-join on EMP. Then use the plus sign (+) as the concatenation operator:

```
1 select a.ename + ' works for ' + b.ename as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

Discussion

The implementation is essentially the same for all the solutions. The difference lies only in the method of string concatenation, and thus one discussion will cover all of the solutions.

The key is the join between MGR and EMPNO. The first step is to build a Cartesian product by joining EMP to itself (only a portion of the rows returned by the Cartesian product is shown here):

```
select a.empno, b.empno
  from emp a, emp b

EMPNO        MGR
----- ----------
 7369       7369
 7369       7499
 7369       7521
 7369       7566
 7369       7654
 7369       7698
 7369       7782
 7369       7788
 7369       7839
 7369       7844
 7369       7876
 7369       7900
 7369       7902
 7369       7934
 7499       7369
 7499       7499
 7499       7521
 7499       7566
 7499       7654
 7499       7698
 7499       7782
```

```
 7499       7788
 7499       7839
 7499       7844
 7499       7876
 7499       7900
 7499       7902
 7499       7934
```

As you can see, by using a Cartesian product you are returning every possible EMPNO/EMPNO combination (such that it looks like the manager for EMPNO 7369 is all the other employees in the table, including EMPNO 7369).

The next step is to filter the results such that you return only each employee and their manager's EMPNO. Accomplish this by joining on MGR and EMPNO:

```
1 select a.empno, b.empno mgr
2   from emp a, emp b
3  where a.mgr = b.empno

     EMPNO        MGR
---------- ----------
      7902       7566
      7788       7566
      7900       7698
      7844       7698
      7654       7698
      7521       7698
      7499       7698
      7934       7782
      7876       7788
      7782       7839
      7698       7839
      7566       7839
      7369       7902
```

Now that you have each employee and the EMPNO of their manager, you can return the name of each manager by simply selecting B.ENAME rather than B.EMPNO. If after some practice you have difficulty grasping how this works, you can use a scalar subquery rather than a self-join to get the answer:

```
select a.ename,
       (select b.ename
          from emp b
         where b.empno = a.mgr) as mgr
  from emp a

ENAME      MGR
---------- ----------
SMITH      FORD
ALLEN      BLAKE
WARD       BLAKE
JONES      KING
```

```
MARTIN     BLAKE
BLAKE      KING
CLARK      KING
SCOTT      JONES
KING
TURNER     BLAKE
ADAMS      SCOTT
JAMES      BLAKE
FORD       JONES
MILLER     CLARK
```

The scalar subquery version is equivalent to the self-join, except for one row: employee KING is in the result set, but that is not the case with the self-join. "Why not?" you might ask. Remember, NULL is never equal to anything, not even itself. In the self-join solution, you use an equi-join between EMPNO and MGR, thus filtering out any employees who have NULL for MGR. To see employee KING when using the self-join method, you must outer join as shown in the following two queries. The first solution uses the ANSI outer join, while the second uses the Oracle outer-join syntax. The output is the same for both and is shown following the second query:

```
/* ANSI */
select a.ename, b.ename mgr
  from emp a left join emp b
    on (a.mgr = b.empno)

/* Oracle */
select a.ename, b.ename mgr
  from emp a, emp b
 where a.mgr = b.empno (+)

ENAME      MGR
---------- ----------
FORD       JONES
SCOTT      JONES
JAMES      BLAKE
TURNER     BLAKE
MARTIN     BLAKE
WARD       BLAKE
ALLEN      BLAKE
MILLER     CLARK
ADAMS      SCOTT
CLARK      KING
BLAKE      KING
JONES      KING
SMITH      FORD
KING
```

13.2 Expressing a Child-Parent-Grandparent Relationship

Problem

Employee CLARK works for KING, and to express that relationship you can use the first recipe in this chapter. What if employee CLARK was in turn a manager for another employee? Consider the following query:

```
select ename,empno,mgr
  from emp
 where ename in ('KING','CLARK','MILLER')

ENAME        EMPNO     MGR
--------- -------- -------
CLARK         7782    7839
KING          7839
MILLER        7934    7782
```

As you can see, employee MILLER works for CLARK who in turn works for KING. You want to express the full hierarchy from MILLER to KING. You want to return the following result set:

```
LEAF___BRANCH___ROOT
--------------------
MILLER-->CLARK-->KING
```

However, the single self-join approach from the previous recipe will not suffice to show the entire relationship from top to bottom. You could write a query that does two self-joins, but what you really need is a general approach for traversing such hierarchies.

Solution

This recipe differs from the first recipe because there is now a three-tier relationship, as the title suggests. If your RDBMS does not supply functionality for traversing tree-structured data, as is the case for Oracle, then you can solve this problem using the CTEs.

DB2, PostgreSQL, and SQL Server

Use the recursive WITH clause to find MILLER's manager, CLARK, and then CLARK's manager, KING. The SQL Server string concatenation operator + is used in this solution:

```
1    with  x (tree,mgr,depth)
2      as  (
3  select  cast(ename as varchar(100)),
4          mgr, 0
5    from  emp
```

```
  where  ename = 'MILLER'
  union  all
select  cast(x.tree+'-->'+e.ename as varchar(100)),
        e.mgr, x.depth+1
  from  emp e, x
 where x.mgr = e.empno
)
select tree leaf___branch___root
  from x
 where depth = 2
```

This solution can work on other databases if the concatenation operator is changed. Hence, change to || for DB2 or CONCAT for PostgreSQL.

MySQL

This is similar to the previous solution, but also needs the RECURSIVE keyword:

```
  with recursive x (tree,mgr,depth)
    as  (
select  cast(ename as varchar(100)),
        mgr, 0
  from  emp
  where  ename = 'MILLER'
  union  all
select  cast(concat(x.tree,'-->',emp.ename) as char(100)),
        e.mgr, x.depth+1
  from  emp e, x
 where x.mgr = e.empno
)
select tree leaf___branch___root
  from x
 where depth = 2
```

Oracle

Use the function SYS_CONNECT_BY_PATH to return MILLER; MILLER's manager, CLARK; and then CLARK's manager, KING. Use the CONNECT BY clause to walk the tree:

```
select ltrim(
         sys_connect_by_path(ename,'-->'),
       '-->') leaf___branch___root
  from emp
 where level = 3
 start with ename = 'MILLER'
connect by prior mgr = empno
```

Discussion

DB2, SQL Server, PostgreSQL, and MySQL

The approach here is to start at the leaf node and walk your way up to the root (as useful practice, try walking in the other direction). The upper part of the UNION ALL simply finds the row for employee MILLER (the leaf node). The lower part of the UNION ALL finds the employee who is MILLER's manager and then finds that person's manager, and this process of finding the "manager's manager" repeats until processing stops at the highest-level manager (the root node). The value for DEPTH starts at 0 and increments automatically by 1 each time a manager is found. DEPTH is a value that DB2 maintains for you when you execute a recursive query.

For an interesting and in-depth introduction to the WITH clause with a focus on its use recursively, see Jonathan Gennick's article "Understanding the WITH Clause".

Next, the second query of the UNION ALL joins the recursive view X to table EMP, to define the parent-child relationship. The query at this point, using SQL Server's concatenation operator, is as follows:

```
  with x (tree,mgr,depth)
    as (
select cast(ename as varchar(100)),
       mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->'+e.ename as varchar(100)),
       e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
)
select tree leaf___branch___root
  from x

TREE            DEPTH
---------- ----------
MILLER               0
CLARK                1
KING                 2
```

At this point, the heart of the problem has been solved; starting from MILLER, return the full hierarchical relationship from bottom to top. What's left then is merely formatting. Since the tree traversal is recursive, simply concatenate the current ENAME from EMP to the one before it, which gives you the following result set:

```
  with x (tree,mgr,depth)
    as (
select  cast(ename as varchar(100)),
        mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->'+e.ename as varchar(100)),
       e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
)
select depth, tree
  from x

DEPTH TREE
----- --------------------------
    0 MILLER
    1 MILLER-->CLARK
    2 MILLER-->CLARK-->KING
```

The final step is to keep only the last row in the hierarchy. There are several ways to do this, but the solution uses DEPTH to determine when the root is reached (obviously, if CLARK has a manager other than KING, the filter on DEPTH would have to change; for a more generic solution that requires no such filter, see the next recipe).

Oracle

The CONNECT BY clause does all the work in the Oracle solution. Starting with MILLER, you walk all the way to KING without the need for any joins. The expression in the CONNECT BY clause defines the relationship of the data and how the tree will be walked:

```
 select ename
   from emp
  start with ename = 'MILLER'
connect by prior mgr = empno

ENAME
--------
MILLER
CLARK
KING
```

The keyword PRIOR lets you access values from the previous record in the hierarchy. Thus, for any given EMPNO, you can use PRIOR MGR to access that employee's manager number. When you see a clause such as CONNECT BY PRIOR MGR = EMPNO, think of that clause as expressing a join between, in this case, parent and child.

For more on CONNECT BY and its use in hierarchical queries, "Hierarchical Queries in Oracle" is a good overview.

At this point, you have successfully displayed the full hierarchy starting from MILLER and ending at KING. The problem is for the most part solved. All that remains is the formatting. Use the function SYS_CONNECT_BY_PATH to append each ENAME to the one before it:

```
 select sys_connect_by_path(ename,'-->') tree
   from emp
  start with ename = 'MILLER'
connect by prior mgr = empno

TREE
---------------------------
-->MILLER
-->MILLER-->CLARK
-->MILLER-->CLARK-->KING
```

Because you are interested in only the complete hierarchy, you can filter on the pseudo-column LEVEL (a more generic approach is shown in the next recipe):

```
 select sys_connect_by_path(ename,'-->') tree
   from emp
  where level = 3
  start with ename = 'MILLER'
connect by prior mgr = empno

TREE
---------------------------
-->MILLER-->CLARK-->KING
```

The final step is to use the LTRIM function to remove the leading --> from the result set.

13.3 Creating a Hierarchical View of a Table

Problem

You want to return a result set that describes the hierarchy of an entire table. In the case of the EMP table, employee KING has no manager, so KING is the root node. You want to display, starting from KING, all employees under KING and all employees (if any) under KING's subordinates. Ultimately, you want to return the following result set:

```
EMP_TREE
------------------------------
KING
KING - BLAKE
KING - BLAKE - ALLEN
KING - BLAKE - JAMES
KING - BLAKE - MARTIN
KING - BLAKE - TURNER
KING - BLAKE - WARD
KING - CLARK
KING - CLARK - MILLER
KING - JONES
KING - JONES - FORD
KING - JONES - FORD - SMITH
KING - JONES - SCOTT
KING - JONES - SCOTT - ADAMS
```

Solution

DB2, PostgreSQL, and SQL Server

Use the recursive WITH clause to start building the hierarchy at KING and then ultimately display all the employees. The solution following uses the DB2 concatenation operator (||). SQL Server users use the concatenation operator (+), and MySQL uses the CONCAT function. Other than the concatenation operators, the solution will work as-is on both RDBMSs:

```
 1   with x (ename,empno)
 2      as (
 3  select cast(ename as varchar(100)),empno
 4    from emp
 5   where mgr is null
 6   union all
 7  select cast(x.ename||' - '||e.ename as varchar(100)),
 8         e.empno
 9    from emp e, x
10   where e.mgr = x.empno
11  )
12  select ename as emp_tree
13    from x
14   order by 1
```

MySQL

MySQL also needs the RECURSIVE keyword:

```
 1   with recursive x (ename,empno)
 2      as (
 3  select cast(ename as varchar(100)),empno
 4    from emp
 5   where mgr is null
```

```
 6   union all
 7  select cast(concat(x.ename,' - ',e.ename) as varchar(100)),
 8         e.empno
 9    from emp e, x
10   where e.mgr = x.empno
11  )
12  select ename as emp_tree
13    from x
14   order by 1
```

Oracle

Use the CONNECT BY function to define the hierarchy. Use the SYS_CONNECT_BY_PATH function to format the output accordingly:

```
1  select ltrim(
2           sys_connect_by_path(ename,' - '),
3         ' - ') emp_tree
4    from emp
5    start with mgr is null
6  connect by prior empno=mgr
7    order by 1
```

This solution differs from the previous recipe in that it includes no filter on the LEVEL pseudo-column. Without the filter, all possible trees (where PRIOR EMPNO=MGR) are displayed.

Discussion

DB2, MySQL, PostgreSQL, and SQL Server

The first step is to identify the root row (employee KING) in the upper part of the UNION ALL in the recursive view X. The next step is to find KING's subordinates, and their subordinates if there are any, by joining recursive view X to table EMP. Recursion will continue until you've returned all employees. Without the formatting you see in the final result set, the result set returned by the recursive view X is shown here:

```
with x (ename,empno)
    as (
select cast(ename as varchar(100)),empno
  from emp
 where mgr is null
 union all
select cast(e.ename as varchar(100)),e.empno
  from emp e, x
 where e.mgr = x.empno
 )
 select ename emp_tree
   from x
```

```
EMP_TREE
----------------
KING
JONES
SCOTT
ADAMS
FORD
SMITH
BLAKE
ALLEN
WARD
MARTIN
TURNER
JAMES
CLARK
MILLER
```

All the rows in the hierarchy are returned (which can be useful), but without the formatting you cannot tell who the managers are. By concatenating each employee to her manager, you return more meaningful output. Produce the desired output simply by using the following:

```
cast(x.ename+','+e.ename as varchar(100))
```

in the SELECT clause of the lower portion of the UNION ALL in recursive view X.

The WITH clause is extremely useful in solving this type of problem, because the hierarchy can change (for example, leaf nodes become branch nodes) without any need to modify the query.

Oracle

The CONNECT BY clause returns the rows in the hierarchy. The START WITH clause defines the root row. If you run the solution without SYS_CONNECT_BY_PATH, you can see that the correct rows are returned (which can be useful), but not formatted to express the relationship of the rows:

```
select ename emp_tree
  from emp
 start with mgr is null
connect by prior empno = mgr

EMP_TREE
-----------------
KING
JONES
SCOTT
ADAMS
FORD
SMITH
BLAKE
ALLEN
```

```
WARD
MARTIN
TURNER
JAMES
CLARK
MILLER
```

By using the pseudo-column LEVEL and the function LPAD, you can see the hierarchy more clearly, and you can ultimately see why SYS_CONNECT_BY_PATH returns the results that you see in the desired output shown earlier:

```
select lpad('.',2*level,'.')||ename emp_tree
   from emp
  start with mgr is null
connect by prior empno = mgr

EMP_TREE
-----------------
..KING
....JONES
......SCOTT
........ADAMS
......FORD
........SMITH
....BLAKE
......ALLEN
......WARD
......MARTIN
......TURNER
......JAMES
....CLARK
......MILLER
```

The indentation in this output indicates who the managers are by nesting subordinates under their superiors. For example, KING works for no one. JONES works for KING. SCOTT works for JONES. ADAMS works for SCOTT.

If you look at the corresponding rows from the solution when using SYS_CONNECT_BY_PATH, you will see that SYS_CONNECT_BY_PATH rolls up the hierarchy for you. When you get to a new node, you see all the prior nodes as well:

```
KING
KING - JONES
KING - JONES - SCOTT
KING - JONES - SCOTT - ADAMS
```

13.4 Finding All Child Rows for a Given Parent Row

Problem

You want to find all the employees who work for JONES, either directly or indirectly (i.e., they work for someone who works for JONES). The list of employees under JONES is shown here (JONES is included in the result set):

```
ENAME
----------
JONES
SCOTT
ADAMS
FORD
SMITH
```

Solution

Being able to move to the absolute top or bottom of a tree is extremely useful. For this solution, there is no special formatting necessary. The goal is to simply return all employees who work under employee JONES, including JONES himself. This type of query really shows the usefulness of recursive SQL extensions like Oracle's CONNECT BY and SQL Server's/DB2's WITH clause.

DB2, PostgreSQL, and SQL Server

Use the recursive WITH clause to find all employees under JONES. Begin with JONES by specifying WHERE ENAME = JONES in the first of the two union queries:

```
 1   with x (ename,empno)
 2     as (
 3 select ename,empno
 4   from emp
 5  where ename = 'JONES'
 6  union all
 7 select e.ename, e.empno
 8   from emp e, x
 9  where x.empno = e.mgr
10 )
11 select ename
12   from x
```

Oracle

Use the CONNECT BY clause and specify START WITH ENAME = *JONES* to find all the employees under JONES:

```
1 select ename
2   from emp
```

```
3  start with ename = 'JONES'
4 connect by prior empno = mgr
```

Discussion

DB2, MySQL, PostgreSQL, and SQL Server

The recursive WITH clause makes this a relatively easy problem to solve. The first part of the WITH clause, the upper part of the UNION ALL, returns the row for employee JONES. You need to return ENAME to see the name and EMPNO so you can use it to join on. The lower part of the UNION ALL recursively joins EMP.MGR to X.EMPNO. The join condition will be applied until the result set is exhausted.

Oracle

The START WTH clause tells the query to make JONES the root node. The condition in the CONNECT BY clause drives the tree walk and will run until the condition is no longer true.

13.5 Determining Which Rows Are Leaf, Branch, or Root Nodes

Problem

You want to determine what type of node a given row is: a leaf, branch, or root. For this example, a leaf node is an employee who is not a manager. A branch node is an employee who is both a manager and also has a manager. A root node is an employee without a manager. You want to return 1 (TRUE) or 0 (FALSE) to reflect the status of each row in the hierarchy. You want to return the following result set:

```
ENAME          IS_LEAF   IS_BRANCH     IS_ROOT
----------  ----------  ----------  ----------
KING                 0           0           1
JONES                0           1           0
SCOTT                0           1           0
FORD                 0           1           0
CLARK                0           1           0
BLAKE                0           1           0
ADAMS                1           0           0
MILLER               1           0           0
JAMES                1           0           0
TURNER               1           0           0
ALLEN                1           0           0
WARD                 1           0           0
MARTIN               1           0           0
SMITH                1           0           0
```

Solution

It is important to realize that the EMP table is modeled in a tree hierarchy, not a recursive hierarchy, and the value for MGR for root nodes is NULL. If EMP were modeled to use a recursive hierarchy, root nodes would be self-referencing (i.e., the value for MGR for employee KING would be KING's EMPNO). We find self-referencing to be counterintuitive and thus are using NULL values for root nodes' MGR. For Oracle users using CONNECT BY and DB2/SQL Server users using WITH, you'll find tree hierarchies easier to work with and potentially more efficient than recursive hierarchies. If you are in a situation where you have a recursive hierarchy and are using CONNECT BY or WITH, watch out: you can end up with a loop in your SQL. You need to code around such loops if you are stuck with recursive hierarchies.

DB2, PostgreSQL, MySQL, and SQL Server

Use three scalar subqueries to determine the correct "Boolean" value (either a 1 or a 0) to return for each node type:

```
 1 select e.ename,
 2        (select sign(count(*)) from emp d
 3          where 0 =
 4            (select count(*) from emp f
 5              where f.mgr = e.empno)) as is_leaf,
 6        (select sign(count(*)) from emp d
 7          where d.mgr = e.empno
 8           and e.mgr is not null) as is_branch,
 9        (select sign(count(*)) from emp d
10          where d.empno = e.empno
11            and d.mgr is null) as is_root
12   from emp e
13 order by 4 desc,3 desc
```

Oracle

The scalar subquery solution will work for Oracle as well and should be used if you are on a version of Oracle prior to Oracle Database 10*g*. The following solution highlights built-in functions provided by Oracle (that were introduced in Oracle Database 10*g*) to identify root and leaf rows. The functions are CONNECT_BY_ROOT and CONNECT_BY_ISLEAF, respectively:

```
1  select ename,
2         connect_by_isleaf is_leaf,
3         (select count(*) from emp e
4           where e.mgr = emp.empno
5             and emp.mgr is not null
6             and rownum = 1) is_branch,
7         decode(ename,connect_by_root(ename),1,0) is_root
8    from emp
```

```
 9   start with mgr is null
10 connect by prior empno = mgr
11 order by 4 desc, 3 desc
```

Discussion

DB2, PostgreSQL, MySQL, and SQL Server

This solution simply applies the rules defined in the "Problem" section to determine leaves, branches, and roots. The first step is to determine whether an employee is a leaf node. If the employee is not a manager (no one works under them), then she is a leaf node. The first scalar subquery, IS_LEAF, is shown here:

```
select e.ename,
       (select sign(count(*)) from emp d
         where 0 =
           (select count(*) from emp f
             where f.mgr = e.empno)) as is_leaf
  from emp e
order by 2 desc

ENAME        IS_LEAF
----------- --------
SMITH              1
ALLEN              1
WARD               1
ADAMS              1
TURNER             1
MARTIN             1
JAMES              1
MILLER             1
JONES              0
BLAKE              0
CLARK              0
FORD               0
SCOTT              0
KING               0
```

Because the output for IS_LEAF should be a 0 or 1, it is necessary to take the SIGN of the COUNT(*) operation. Otherwise, you would get 14 instead of 1 for leaf rows. As an alternative, you can use a table with only one row to count against, because you only want to return 0 or 1. For example:

```
select e.ename,
       (select count(*) from t1 d
         where not exists
           (select null from emp f
             where f.mgr = e.empno)) as is_leaf
  from emp e
order by 2 desc
```

```
ENAME         IS_LEAF
---------- ----------
SMITH              1
ALLEN              1
WARD               1
ADAMS              1
TURNER             1
MARTIN             1
JAMES              1
MILLER             1
JONES              0
BLAKE              0
CLARK              0
FORD               0
SCOTT              0
KING               0
```

The next step is to find branch nodes. If an employee is a manager (someone works for them) and they also happen to work for someone else, then the employee is a branch node. The results of the scalar subquery IS_BRANCH are shown here:

```
select e.ename,
       (select sign(count(*)) from emp d
         where d.mgr = e.empno
          and e.mgr is not null) as is_branch
  from emp e
order by 2 desc


ENAME       IS_BRANCH
----------- ---------
JONES               1
BLAKE               1
SCOTT               1
CLARK               1
FORD                1
SMITH               0
TURNER              0
MILLER              0
JAMES               0
ADAMS               0
KING                0
ALLEN               0
MARTIN              0
WARD                0
```

Again, it is necessary to take the SIGN of the COUNT(*) operation. Otherwise, you will get (potentially) values greater than 1 when a node is a branch. Like scalar subquery IS_LEAF, you can use a table with one row to avoid using SIGN. The following solution uses the T1 table:

```
select e.ename,
      (select count(*) from t1 t
        where exists (
         select null from emp f
          where f.mgr = e.empno
            and e.mgr is not null)) as is_branch
  from emp e
order by 2 desc
```

```
ENAME            IS_BRANCH
--------------- ----------
JONES                    1
BLAKE                    1
SCOTT                    1
CLARK                    1
FORD                     1
SMITH                    0
TURNER                   0
MILLER                   0
JAMES                    0
ADAMS                    0
KING                     0
ALLEN                    0
MARTIN                   0
WARD                     0
```

The last step is to find the root nodes. A root node is defined as an employee who is a manager but who does not work for anyone else. In table EMP, only KING is a root node. Scalar subquery IS_ROOT is shown here:

```
select e.ename,
       (select sign(count(*)) from emp d
         where d.empno = e.empno
           and d.mgr is null) as is_root
  from emp e
order by 2 desc
```

```
ENAME         IS_ROOT
----------  ---------
KING                1
SMITH               0
ALLEN               0
WARD                0
JONES               0
TURNER              0
JAMES               0
MILLER              0
FORD                0
ADAMS               0
MARTIN              0
```

```
BLAKE               0
CLARK               0
SCOTT               0
```

Because EMP is a small 14-row table, it is easy to see that employee KING is the only root node, so in this case taking the SIGN of the COUNT(*) operation is not strictly necessary. If there can be multiple root nodes, then you can use SIGN, or you can use a one-row table in the scalar subquery as is shown earlier for IS_BRANCH and IS_LEAF.

Oracle

For those of you on versions of Oracle prior to Oracle Database 10*g*, you can follow the discussion for the other RDBMSs, as that solution will work (without modifications) in Oracle. If you are on Oracle Database 10*g* or later, you may want to take advantage of two functions to make identifying root and leaf nodes a simple task: they are CONNECT_BY_ROOT and CONNECT_BY_ISLEAF, respectively. As of the time of this writing, it is necessary to use CONNECT BY in your SQL statement in order for you to be able to use CONNECT_BY_ROOT and CONNECT_BY_ISLEAF. The first step is to find the leaf nodes by using CONNECT_BY_ISLEAF as follows:

```
select ename,
       connect_by_isleaf is_leaf
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc


ENAME          IS_LEAF
----------  ----------
ADAMS                1
SMITH                1
ALLEN                1
TURNER               1
MARTIN               1
WARD                 1
JAMES                1
MILLER               1
KING                 0
JONES                0
BLAKE                0
CLARK                0
FORD                 0
SCOTT                0
```

The next step is to use a scalar subquery to find the branch nodes. Branch nodes are employees who are managers but who also work for someone else:

```
select ename,
        (select count(*) from emp e
          where e.mgr = emp.empno
            and emp.mgr is not null
            and rownum = 1) is_branch
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

ENAME       IS_BRANCH
---------- ----------
JONES               1
SCOTT               1
BLAKE               1
FORD                1
CLARK               1
KING                0
MARTIN              0
MILLER              0
JAMES               0
TURNER              0
WARD                0
ADAMS               0
ALLEN               0
SMITH               0
```

The filter on ROWNUM is necessary to ensure that you return a count of 1 or 0, and nothing else.

The last step is to identify the root nodes by using the function CONNECT_BY_ROOT. The solution finds the ENAME for the root node and compares it with all the rows returned by the query. If there is a match, that row is the root node:

```
select ename,
        decode(ename,connect_by_root(ename),1,0) is_root
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

ENAME          IS_ROOT
----------  ----------
KING                 1
JONES                0
SCOTT                0
ADAMS                0
FORD                 0
SMITH                0
BLAKE                0
ALLEN                0
WARD                 0
```

```
MARTIN               0
TURNER               0
JAMES                0
CLARK                0
MILLER               0
```

The SYS_CONNECT_BY_PATH function rolls up a hierarchy starting from the root value, as shown here:

```
select ename,
       ltrim(sys_connect_by_path(ename,','),',') path
  from emp
start with mgr is null
connect by prior empno=mgr

ENAME      PATH
---------- ----------------------------
KING       KING
JONES      KING,JONES
SCOTT      KING,JONES,SCOTT
ADAMS      KING,JONES,SCOTT,ADAMS
FORD       KING,JONES,FORD
SMITH      KING,JONES,FORD,SMITH
BLAKE      KING,BLAKE
ALLEN      KING,BLAKE,ALLEN
WARD       KING,BLAKE,WARD
MARTIN     KING,BLAKE,MARTIN
TURNER     KING,BLAKE,TURNER
JAMES      KING,BLAKE,JAMES
CLARK      KING,CLARK
MILLER     KING,CLARK,MILLER
```

To get the root row, simply substring out the first ENAME in PATH:

```
select ename,
       substr(root,1,instr(root,',')-1) root
  from (
select ename,
       ltrim(sys_connect_by_path(ename,','),',') root
  from emp
start with mgr is null
connect by prior empno=mgr
       )

ENAME      ROOT
---------- ----------
KING
JONES      KING
SCOTT      KING
ADAMS      KING
FORD       KING
SMITH      KING
BLAKE      KING
```

```
ALLEN      KING
WARD       KING
MARTIN     KING
TURNER     KING
JAMES      KING
CLARK      KING
MILLER     KING
```

The last step is to flag the result from the ROOT column; if it is NULL, that is your root row.

13.6 Summing Up

The spread of CTEs across all vendors has made standardized approaches to hierarchical queries far more achievable. This a great step forward as hierarchical relationships appear in many kinds of data, even data where the relationship isn't necessarily planned for, so queries need to account for it.

CHAPTER 14

Odds 'n' Ends

This chapter contains queries that didn't fit in any other chapter, either because the chapter they would belong to is already long enough, or because the problems they solve are more fun than realistic. This chapter is meant to be a "fun" chapter, in that the recipes here may or may not be recipes that you would actually use; nevertheless, the queries are interesting, and we wanted to include them in this book.

14.1 Creating Cross-Tab Reports Using SQL Server's PIVOT Operator

Problem

You want to create a cross-tab report to transform your result set's rows into columns. You are aware of traditional methods of pivoting but would like to try something different. In particular, you want to return the following result set without using CASE expressions or joins:

```
DEPT_10     DEPT_20     DEPT_30    DEPT_40
------- ----------- ----------- ----------
      3           5           6          0
```

Solution

Use the PIVOT operator to create the required result set without CASE expressions or additional joins:

```
1 select [10] as dept_10,
2        [20] as dept_20,
3        [30] as dept_30,
4        [40] as dept_40
5   from (select deptno, empno from emp) driver
```

```
6  pivot (
7     count(driver.empno)
8     for driver.deptno in ( [10],[20],[30],[40] )
9  ) as empPivot
```

Discussion

The PIVOT operator may seem strange at first, but the operation it performs in the solution is technically the same as the more familiar transposition query shown here:

```
select sum(case deptno when 10 then 1 else 0 end) as dept_10,
       sum(case deptno when 20 then 1 else 0 end) as dept_20,
       sum(case deptno when 30 then 1 else 0 end) as dept_30,
       sum(case deptno when 40 then 1 else 0 end) as dept_40
  from emp

DEPT_10    DEPT_20    DEPT_30    DEPT_40
------- ---------- ---------- ----------
      3          5          6          0
```

Now that you know what is essentially happening, let's break down what the PIVOT operator is doing. Line 5 of the solution shows an inline view named DRIVER:

```
from (select deptno, empno from emp) driver
```

We've used the alias DRIVER because the rows from this inline view (or table expression) feed directly into the PIVOT operation. The PIVOT operator rotates the rows to columns by evaluating the items listed on line 8 in the FOR list (shown here):

```
for driver.deptno in ( [10],[20],[30],[40] )
```

The evaluation goes something like this:

1. If there are any DEPTNOs with a value of 10, perform the aggregate operation defined (COUNT(DRIVER.EMPNO)) for those rows.
2. Repeat for DEPTNOs 20, 30, and 40.

The items listed in the brackets on line 8 serve not only to define values for which aggregation is performed; the items also become the column names in the result set (without the square brackets). In the SELECT clause of the solution, the items in the FOR list are referenced and aliased. If you do not alias the items in the FOR list, the column names become the items in the FOR list sans brackets.

Interestingly enough, since inline view DRIVER is just that—an inline view—you may put more complex SQL in there. For example, consider the situation where you want to modify the result set such that the actual department name is the name of the column. Listed here are the rows in table DEPT:

```
select * from dept

DEPTNO DNAME          LOC
------ -------------- -------------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

You want to use PIVOT to return the following result set:

```
ACCOUNTING   RESEARCH      SALES OPERATIONS
---------- ---------- ---------- ----------
         3          5          6          0
```

Because inline view DRIVER can be practically any valid table expression, you can perform the join from table EMP to table DEPT and then have PIVOT evaluate those rows. The following query will return the desired result set:

```
select [ACCOUNTING] as ACCOUNTING,
       [SALES]      as SALES,
       [RESEARCH]   as RESEARCH,
       [OPERATIONS] as OPERATIONS
  from (
          select d.dname, e.empno
            from emp e,dept d
           where e.deptno=d.deptno

         ) driver
  pivot (
   count(driver.empno)
   for driver.dname in ([ACCOUNTING],[SALES],[RESEARCH],[OPERATIONS])
  ) as empPivot
```

As you can see, PIVOT provides an interesting spin on pivoting result sets. Regardless of whether you prefer using it to the traditional methods of pivoting, it's nice to have another tool in your toolbox.

14.2 Unpivoting a Cross-Tab Report Using SQL Server's UNPIVOT Operator

Problem

You have a pivoted result set (or simply a fact table), and you want to unpivot the result set. For example, instead of having a result set with one row and four columns, you want to return a result set with two columns and four rows. Using the result set from the previous recipe, you want to convert it from this:

```
ACCOUNTING   RESEARCH      SALES OPERATIONS
---------- ---------- ---------- ----------
         3          5          6          0
```

to this:

```
DNAME                 CNT
------------- ----------
ACCOUNTING              3
RESEARCH                5
SALES                   6
OPERATIONS              0
```

Solution

You didn't think SQL Server would give you the ability to PIVOT without being able to UNPIVOT, did you? To unpivot the result set, just use it as the driver and let the UNPIVOT operator do all the work. All you need to do is specify the column names:

```
 1   select DNAME, CNT
 2     from (
 3       select [ACCOUNTING] as ACCOUNTING,
 4              [SALES]      as SALES,
 5              [RESEARCH]   as RESEARCH,
 6              [OPERATIONS] as OPERATIONS
 7         from (
 8                select d.dname, e.empno
 9                  from emp e,dept d
10                 where e.deptno=d.deptno
11
12               ) driver
13         pivot (
14           count(driver.empno)
15           for driver.dname in ([ACCOUNTING],[SALES],[RESEARCH],[OPERATIONS])
16         ) as empPivot
17  )  new_driver
18  unpivot (cnt for dname in (ACCOUNTING,SALES,RESEARCH,OPERATIONS)
19  ) as un_pivot
```

Ideally, before reading this recipe you've read the one prior to it, because the inline view NEW_DRIVER is simply the code from the previous recipe (if you don't understand it, please refer to the previous recipe before looking at this one). Since lines 3–16 consist of code you've already seen, the only new syntax is on line 18, where you use UNPIVOT.

The UNPIVOT command simply looks at the result set from NEW_DRIVER and evaluates each column and row. For example, the UNPIVOT operator evaluates the column names from NEW_DRIVER. When it encounters ACCOUNTING, it transforms the column name ACCOUNTING into a row value (under the column DNAME). It also takes the value for ACCOUNTING from NEW_DRIVER (which is 3) and returns that as part of the ACCOUNTING row as well (under the column

CNT). UNPIVOT does this for each of the items specified in the FOR list and simply returns each one as a row.

The new result set is now skinny and has two columns, DNAME and CNT, with four rows:

```
select DNAME, CNT
  from (
    select [ACCOUNTING] as ACCOUNTING,
           [SALES]      as SALES,
           [RESEARCH]   as RESEARCH,
           [OPERATIONS] as OPERATIONS
      from (
              select d.dname, e.empno
                from emp e,dept d
               where e.deptno=d.deptno

           )  driver
     pivot (
       count(driver.empno)
       for driver.dname in ( [ACCOUNTING],[SALES],[RESEARCH],[OPERATIONS] )
     ) as empPivot
) new_driver
unpivot (cnt for dname in (ACCOUNTING,SALES,RESEARCH,OPERATIONS)
) as un_pivot

DNAME                 CNT
-------------- ----------
ACCOUNTING             3
RESEARCH               5
SALES                  6
OPERATIONS             0
```

14.3 Transposing a Result Set Using Oracle's MODEL Clause

Problem

Like the first recipe in this chapter, you want to find an alternative to the traditional pivoting techniques you've seen already. You want to try your hand at Oracle's MODEL clause. Unlike SQL Server's PIVOT operator, Oracle's MODEL clause does not exist to transpose result sets; as a matter of fact, it would be quite accurate to say the application of the MODEL clause for pivoting would be a misuse and clearly not what the MODEL clause was intended for. Nevertheless, the MODEL clause provides for an interesting approach to a common problem. For this particular problem, you want to transform the following result set from this:

```
select deptno, count(*) cnt
  from emp
 group by deptno

DEPTNO        CNT
------ ----------
    10          3
    20          5
    30          6
```

to this:

```
       D10        D20        D30
---------- ---------- ----------
         3          5          6
```

Solution

Use aggregation and CASE expressions in the MODEL clause just as you would use them if pivoting with traditional techniques. The main difference in this case is that you use arrays to store the values of the aggregation and return the arrays in the result set:

```
select max(d10) d10,
       max(d20) d20,
       max(d30) d30
  from (
select d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )
  )
```

Discussion

The MODEL clause is a powerful addition to the Oracle SQL toolbox. Once you begin working with MODEL, you'll notice helpful features such as iteration, array access to row values, the ability to "upsert" rows into a result set, and the ability to build reference models. You'll quickly see that this recipe doesn't take advantage of any of the cool features the MODEL clause offers, but it's nice to be able to look at a problem from multiple angles and use different features in unexpected ways (if for no other reason than to learn where certain features are more useful than others).

The first step to understanding the solution is to examine the inline view in the FROM clause. The inline view simply counts the number of employees in each DEPTNO in table EMP. The results are shown here:

```
select deptno, count(*) cnt
  from emp
 group by deptno

DEPTNO        CNT
------ ----------
    10          3
    20          5
    30          6
```

This result set is what is given to MODEL to work with. Examining the MODEL clause, you see three subclauses that stand out: DIMENSION BY, MEASURES, and RULES. Let's start with MEASURES.

The items in the MEASURES list are simply the arrays you are declaring for this query. The query uses four arrays: DEPTNO, D10, D20, and D30. Like columns in a SELECT list, arrays in the MEASURES list can have aliases. As you can see, three of the four arrays are actually CNT from the inline view.

If the MEASURES list contains our arrays, then the items in the DIMENSION BY subclause are the array indices. Consider this: array D10 is simply an alias for CNT. If you look at the result set for the previous inline view, you'll see that CNT has three values: 3, 5, and 6. When you create an array of CNT, you are creating an array with three elements, namely, the three integers: 3, 5, and 6. Now, how do you access these values from the array individually? You use the array index. The index, defined in the DIMENSION BY subclause, has the values of 10, 20, and 30 (from the result set above). So, for example, the following expression:

```
d10[10]
```

would evaluate to 3, as you are accessing the value for CNT in array D10 for DEPTNO 10 (which is 3).

Because all three arrays (D10, D20, D30) contain the values from CNT, all three of them have the same results. How then do we get the proper count into the correct array? Enter the RULES subclause. If you look at the result set for the inline view shown earlier, you'll see that the values for DEPTNO are 10, 20, and 30. The expressions involving CASE in the RULES clause simply evaluate each value in the DEPTNO array:

- If the value is 10, store the CNT for DEPTNO 10 in D10[10] or else store 0.
- If the value is 20, store the CNT for DEPTNO 20 in D20[20] or else store 0.
- If the value is 30, store the CNT for DEPTNO 30 in D30[30] or else store 0.

If you find yourself feeling a bit like Alice tumbling down the rabbit hole, don't worry; just stop and execute what's been discussed thus far. The following result set represents what has been discussed. Sometimes it's easier to read a bit, look at the code that actually performs what you just read, and then go back and read it again. The following is quite simple once you see it in action:

```
select deptno, d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )

  DEPTNO        D10        D20        D30
  ------ ---------- ---------- ----------
      10          3          0          0
      20          0          5          0
      30          0          0          6
```

As you can see, the RULES subclause is what changed the values in each array. If you are still not catching on, simply execute the same query but comment out the expressions in the RULES subclass:

```
select deptno, d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
    /*
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
    */
  )

  DEPTNO        D10        D20        D30
  ------ ---------- ---------- ----------
      10          3          3          3
      20          5          5          5
      30          6          6          6
```

It should be clear now that the result set from the MODEL clause is the same as the inline view, except that the COUNT operation is aliased D10, D20, and D30. The following query proves this:

```
select deptno, count(*) d10, count(*) d20, count(*) d30
  from emp
 group by deptno

 DEPTNO        D10        D20        D30
 ------ ---------- ---------- ----------
     10          3          3          3
     20          5          5          5
     30          6          6          6
```

So, all the MODEL clause did was to take the values for DEPTNO and CNT, put them into arrays, and then make sure that each array represents a single DEPTNO. At this point, arrays D10, D20, and D30 each have a single nonzero value representing the CNT for a given DEPTNO. The result set is already transposed, and all that is left is to use the aggregate function MAX (you could have used MIN or SUM; it would make no difference in this case) to return only one row:

```
select max(d10) d10,
       max(d20) d20,
       max(d30) d30
  from (
select d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )
  )

        D10        D20        D30
  ---------- ---------- ----------
           3          5          6
```

14.4 Extracting Elements of a String from Unfixed Locations

Problem

You have a string field that contains serialized log data. You want to parse through the string and extract the relevant information. Unfortunately, the relevant information is not at fixed points in the string. Instead, you must use the fact that certain characters exist around the information you need, to extract said information. For example, consider the following strings:

```
xxxxxabc[867]xxx[-]xxxx[5309]xxxxx
xxxxxtime:[11271978]favnum:[4]id:[Joe]xxxxx
call:[F_GET_ROWS()]b1:[ROSEWOOD…SIR]b2:[44400002]77.90xxxxx
film:[non_marked]qq:[unit]tailpipe:[withabanana?]80sxxxxx
```

You want to extract the values between the square brackets, returning the following result set:

```
FIRST_VAL       SECOND_VAL          LAST_VAL
--------------- ------------------- ---------------
867             -                   5309
11271978        4                   Joe
F_GET_ROWS()   ROSEWOOD…SIR 44400002
non_marked      unit                withabanana?
```

Solution

Despite not knowing the exact locations within the string of the interesting values, you do know that they are located between square brackets [], and you know there are three of them. Use Oracle's built-in function INSTR to find the locations of the brackets. Use the built-in function SUBSTR to extract the values from the string. View V will contain the strings to parse and is defined as follows (its use is strictly for readability):

```
create view V
as
select 'xxxxxabc[867]xxx[-]xxxx[5309]xxxxx' msg
    from dual
   union all
  select 'xxxxxtime:[11271978]favnum:[4]id:[Joe]xxxxx' msg
    from dual
   union all
  select 'call:[F_GET_ROWS()]b1:[ROSEWOOD…SIR]b2:[44400002]77.90xxxxx' msg
    from dual
   union all
  select 'film:[non_marked]qq:[unit]tailpipe:[withabanana?]80sxxxxx' msg
    from dual

 1 select substr(msg,
 2         instr(msg,'[',1,1)+1,
 3         instr(msg,']',1,1)-instr(msg,'[',1,1)-1) first_val,
 4        substr(msg,
 5         instr(msg,'[',1,2)+1,
 6         instr(msg,']',1,2)-instr(msg,'[',1,2)-1) second_val,
 7        substr(msg,
 8         instr(msg,'[',-1,1)+1,
 9         instr(msg,']',-1,1)-instr(msg,'[',-1,1)-1) last_val
10  from V
```

Discussion

Using Oracle's built-in function INSTR makes this problem fairly simple to solve. Since you know the values you are after are enclosed in [], and that there are three sets of [], the first step to this solution is to simply use INSTR to find the numeric positions of [] in each string. The following example returns the numeric position of the opening and closing brackets in each row:

```
select instr(msg,'[',1,1) "1st_[",
                 instr(msg,']',1,1) "]_1st",
                 instr(msg,'[',1,2) "2nd_[",
                 instr(msg,']',1,2) "]_2nd",
                 instr(msg,'[',-1,1) "3rd_[",
                 instr(msg,']',-1,1) "]_3rd"
            from V

  1st_[ ]_1st      2nd_[ ]_2nd      3rd_[ ]_3rd
  ------ ----- ---------- ----- ---------- -----
       9    13         17    19         24    29
      11    20         28    30         34    38
       6    19         23    38         42    51
       6    17         21    26         36    49
```

At this point, the hard work is done. All that is left is to plug the numeric positions into SUBSTR to parse MSG at those locations. You'll notice that in the complete solution there's some simple arithmetic on the values returned by INSTR, particularly, +1 and –1; this is necessary to ensure the opening square bracket, [, is not returned in the final result set. Listed here is the solution less addition and subtraction of 1 on the return values from INSTR; notice how each value has a leading square bracket:

```
select substr(msg,
       instr(msg,'[',1,1),
       instr(msg,']',1,1)-instr(msg,'[',1,1)) first_val,
      substr(msg,
       instr(msg,'[',1,2),
       instr(msg,']',1,2)-instr(msg,'[',1,2)) second_val,
      substr(msg,
       instr(msg,'[',-1,1),
       instr(msg,']',-1,1)-instr(msg,'[',-1,1)) last_val
  from V

FIRST_VAL       SECOND_VAL           LAST_VAL
--------------- -------------------- -------
[867            [-                   [5309
[11271978       [4                   [Joe
[F_GET_ROWS()  [ROSEWOOD…SIR [44400002
[non_marked     [unit                [withabanana?
```

From the previous result set, you can see that the open bracket is there. You may be thinking: "OK, put the addition of 1 to INSTR back and the leading square bracket goes away. Why do we need to subtract 1?" The reason is this: if you put the addition

back but leave out the subtraction, you end up including the closing square bracket, as shown here:

```
select substr(msg,
        instr(msg,'[',1,1)+1,
        instr(msg,']',1,1)-instr(msg,'[',1,1)) first_val,
   substr(msg,
    instr(msg,'[',1,2)+1,
    instr(msg,']',1,2)-instr(msg,'[',1,2)) second_val,
    substr(msg,
    instr(msg,'[',-1,1)+1,
    instr(msg,']',-1,1)-instr(msg,'[',-1,1)) last_val
  from V

FIRST_VAL       SECOND_VAL           LAST_VAL
--------------- ---------------      -------------
867]            -]                   5309]
11271978]       4]                   Joe]
F_GET_ROWS()]  ROSEWOOD…SIR]  44400002]
non_marked]     unit]                withabanana?]
```

At this point it should be clear: to ensure you include neither of the square brackets, you must add one to the beginning index and subtract one from the ending index.

14.5 Finding the Number of Days in a Year (an Alternate Solution for Oracle)

Problem

You want to find the number of days in a year.

This recipe presents an alternative solution to "Determining the Number of Days in a Year" from Chapter 9. This solution is specific to Oracle.

Solution

Use the TO_CHAR function to format the last date of the year into a three-digit day-of-the-year number:

```
1 select 'Days in 2021: '||
2        to_char(add_months(trunc(sysdate,'y'),12)-1,'DDD')
3        as report
4   from dual
5 union all
6 select 'Days in 2020: '||
7        to_char(add_months(trunc(
```

```
8        to_date('01-SEP-2020'),'y'),12)-1,'DDD')
9   from dual

REPORT
-----------------
Days in 2021: 365
Days in 2020: 366
```

Discussion

Begin by using the TRUNC function to return the first day of the year for the given date, as follows:

```
select trunc(to_date('01-SEP-2020'),'y')
  from dual

TRUNC(TO_DA
-----------
01-JAN-2020
```

Next, use ADD_MONTHS to add one year (12 months) to the truncated date. Then subtract one day, bringing you to the end of the year in which your original date falls:

```
select add_months(
         trunc(to_date('01-SEP-2020'),'y'),
         12) before_subtraction,
       add_months(
         trunc(to_date('01-SEP-2020'),'y'),
         12)-1 after_subtraction
  from dual

BEFORE_SUBT AFTER_SUBTR
----------- -----------
01-JAN-2021 31-DEC-2020
```

Now that you have found the last day in the year you are working with, simply use TO_CHAR to return a three-digit number representing on which day (1st, 50th, etc.) of the year the last day is:

```
select to_char(
         add_months(
           trunc(to_date('01-SEP-2020'),'y'),
           12)-1,'DDD') num_days_in_2020
  from dual

NUM
---
366
```

14.6 Searching for Mixed Alphanumeric Strings

Problem

You have a column with mixed alphanumeric data. You want to return those rows that have both alphabetical and numeric characters; in other words, if a string has only number or only letters, do not return it. The return values should have a mix of both letters and numbers. Consider the following data:

```
STRINGS
------------
1010 switch
333
3453430278
ClassSummary
findRow 55
threes
```

The final result set should contain only those rows that have both letters and numbers:

```
STRINGS
------------
1010 switch
findRow 55
```

Solution

Use the built-in function TRANSLATE to convert each occurrence of a letter or digit into a specific character. Then keep only those strings that have at least one occurrence of both. The solution uses Oracle syntax, but both DB2 and PostgreSQL support TRANSLATE, so modifying the solution to work on those platforms should be trivial:

```
with v as (
select 'ClassSummary' strings from dual union
select '3453430278'           from dual union
select 'findRow 55'           from dual union
select '1010 switch'          from dual union
select '333'                  from dual union
select 'threes'               from dual
)
select strings
  from (
select strings,
translate(
strings,
'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
from v
```

```
) x
whereinstr(translated,'#') > 0
and instr(translated,'*') > 0
```

As an alternative to the WITH clause, you may use an inline view or simply create a view.

Discussion

The TRANSLATE function makes this problem extremely easy to solve. The first step is to use TRANSLATE to identify all letters and all digits by pound (#) and asterisk (*) characters, respectively. The intermediate results (from inline view X) are as follows:

```
with v as (
select 'ClassSummary' strings from dual union
select '3453430278'           from dual union
select 'findRow 55'           from dual union
select '1010 switch'          from dual union
select '333'                  from dual union
select 'threes'               from dual
)
select strings,
       translate(
         strings,
         'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v

STRINGS       TRANSLATED
------------- -----------
1010 switch   **** ######
333           ***
3453430278    **********
ClassSummary  C####S######
findRow 55    ####R## **
threes        ######
```

At this point, it is only a matter of keeping those rows that have at least one instance each of # and *. Use the function INSTR to determine whether # and * are in a string. If those two characters are, in fact, present, then the value returned will be greater than zero. The final strings to return, along with their translated values, are shown next for clarity:

```
with v as (
select 'ClassSummary' strings from dual union
select '3453430278'           from dual union
select 'findRow 55'           from dual union
```

```
select '1010 switch'          from dual union
select '333'                  from dual union
select 'threes'               from dual
)
select strings, translated
  from (
select strings,
       translate(
         strings,
         'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v
       )
 where instr(translated,'#') > 0
   and instr(translated,'*') > 0

STRINGS      TRANSLATED
------------ ------------
1010 switch  **** ######
findRow 55   ####R## **
```

14.7 Converting Whole Numbers to Binary Using Oracle

Problem

You want to convert a whole number to its binary representation on an Oracle system. For example, you would like to return all the salaries in table EMP in binary as part of the following result set:

```
ENAME        SAL SAL_BINARY
---------- ----- --------------------
SMITH        800 1100100000
ALLEN       1600 11001000000
WARD        1250 10011100010
JONES       2975 101110011111
MARTIN      1250 10011100010
BLAKE       2850 101100100010
CLARK       2450 100110010010
SCOTT       3000 101110111000
KING        5000 1001110001000
TURNER      1500 10111011100
ADAMS       1100 10001001100
JAMES        950 1110110110
FORD        3000 101110111000
MILLER      1300 10100010100
```

Solution

Because of MODEL's ability to iterate and provide array access to row values, it is a natural choice for this operation (assuming you are forced to solve the problem in

SQL, as a stored function is more appropriate here). Like the rest of the solutions in this book, even if you don't find a practical application for this code, focus on the technique. It is useful to know that the MODEL clause can perform procedural tasks while still keeping SQL's set-based nature and power. So, even if you find yourself saying, "I'd never do this in SQL," that's fine. We're in no way suggesting you should or shouldn't. We remind you to focus on the technique, so you can apply it to whatever you consider a more "practical" application.

The following solution returns all ENAME and SAL from table EMP, while calling the MODEL clause in a scalar subquery (this way it serves as sort of a standalone function from table EMP that simply receives an input, processes it, and returns a value, much like a function would):

```
 1 select ename,
 2        sal,
 3        (
 4        select bin
 5          from dual
 6         model
 7         dimension by ( 0 attr )
 8         measures ( sal num,
 9                    cast(null as varchar2(30)) bin,
10                   '0123456789ABCDEF' hex
11                  )
12         rules iterate (10000) until (num[0] <= 0) (
13           bin[0] = substr(hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
14           num[0] = trunc(num[cv()]/2)
15         )
16        ) sal_binary
17   from emp
```

Discussion

We mentioned in the "Solution" section that this problem is most likely better solved via a stored function. Indeed, the idea for this recipe came from a function. As a matter of fact, this recipe is an adaptation of a function called TO_BASE, written by Tom Kyte of Oracle Corporation. Like other recipes in this book that you may decide not to use, even if you do not use this recipe, it does a nice job of showing of some of the features of the MODEL clause such as iteration and array access of rows.

To make the explanation easier, we focus on a slight variation of the subquery containing the MODEL clause. The code that follows is essentially the subquery from the solution, except that it's been hardwired to return the value 2 in binary:

```
select bin
  from dual
 model
 dimension by ( 0 attr )
 measures ( 2 num,
```

```
            cast(null as varchar2(30)) bin,
            '0123456789ABCDEF' hex
          )
 rules iterate (10000) until (num[0] <= 0) (
   bin[0] = substr (hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
   num[0] = trunc(num[cv()]/2)
 )

BIN
----------
10
```

The following query outputs the values returned from one iteration of the RULES defined in the previous query:

```
select 2 start_val,
       '0123456789ABCDEF' hex,
       substr('0123456789ABCDEF',mod(2,2)+1,1) ||
       cast(null as varchar2(30)) bin,
       trunc(2/2) num
  from dual

START_VAL HEX              BIN        NUM
--------- ---------------- ---------- ---
        2 0123456789ABCDEF 0            1
```

START_VAL represents the number you want to convert to binary, which in this case is 2. The value for BIN is the result of a substring operation on *0123456789ABCDEF* (HEX, in the original solution). The value for NUM is the test that will determine when you exit the loop.

As you can see from the preceding result set, the first time through the loop BIN is 0 and NUM is 1. Because NUM is not less than or equal to 0, another loop iteration occurs. The following SQL statement shows the results of the next iteration:

```
select num start_val,
       substr('0123456789ABCDEF',mod(1,2)+1,1) || bin bin,
       trunc(1/2) num
  from (
select 2 start_val,
       '0123456789ABCDEF' hex,
       substr('0123456789ABCDEF',mod(2,2)+1,1) ||
       cast(null as varchar2(30)) bin,
       trunc(2/2) num
  from dual
       )

START_VAL BIN        NUM
--------- ---------- ---
        1 10           0
```

The next time through the loop, the result of the substring operation on HEX returns 1, and the prior value of BIN, 0, is appended to it. The test, NUM, is now 0; thus, this is the last iteration, and the return value "10" is the binary representation of the number 2. Once you're comfortable with what's going on, you can remove the iteration from the MODEL clause and step through it row by row to follow how the rules are applied to come to the final result set, as is shown here:

```
select 2 orig_val, num, bin
  from dual
 model
 dimension by ( 0 attr )
 measures ( 2 num,
            cast(null as varchar2(30)) bin,
           '0123456789ABCDEF' hex
          )
 rules (
   bin[0] = substr (hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
   num[0] = trunc(num[cv()]/2),
   bin[1] = substr (hex[0],mod(num[0],2)+1,1)||bin[0],
   num[1] = trunc(num[0]/2)
 )

ORIG_VAL NUM BIN
-------- --- ---------
       2   1 0
       2   0 10
```

14.8 Pivoting a Ranked Result Set

Problem

You want to rank the values in a table and then pivot the result set into three columns. The idea is to show the top three, the next three, and then all the rest. For example, you want to rank the employees in table EMP by SAL and then pivot the results into three columns. The desired result set is as follows:

```
TOP_3           NEXT_3          REST
--------------- --------------- --------------
KING    (5000)  BLAKE   (2850)  TURNER (1500)
FORD    (3000)  CLARK   (2450)  MILLER (1300)
SCOTT   (3000)  ALLEN   (1600)  MARTIN (1250)
JONES   (2975)                  WARD   (1250)
                                ADAMS  (1100)
                                JAMES  (950)
                                SMITH  (800)
```

Solution

The key to this solution is to first use the window function DENSE_RANK OVER to rank the employees by SAL while allowing for ties. By using DENSE_RANK OVER, you can easily see the top three salaries, the next three salaries, and then all the rest.

Next, use the window function ROW_NUMBER OVER to rank each employee within their group (the top three, next three, or last group). From there, simply perform a classic transpose, while using the built-in string functions available on your platform to beautify the results. The following solution uses Oracle syntax. Since all vendors now support window functions, converting the solution to work for other platforms is trivial:

```
select max(case grp when 1 then rpad(ename,6) ||
                    ' ('|| sal ||')' end) top_3,
      max(case grp when 2 then rpad(ename,6) ||
                   ' ('|| sal ||')' end) next_3,
      max(case grp when 3 then rpad(ename,6) ||
                    ' ('|| sal ||')' end) rest
  from (
select ename,
       sal,
       rnk,
       case when rnk <= 3 then 1
            when rnk <= 6 then 2
            else                 3
       end grp,
       row_number()over (
         partition by case when rnk <= 3 then 1
                           when rnk <= 6 then 2
                           else                 3
                      end
             order by sal desc, ename
       ) grp_rnk
  from (
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp
       ) x
       ) y
 group by grp_rnk
```

Discussion

This recipe is a perfect example of how much you can accomplish with so little, with the help of window functions. The solution may look involved, but as you break it down from inside out, you will be surprised how simple it is. Let's begin by executing inline view X first:

```
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp

ENAME        SAL        RNK
---------- ----- ----------
KING        5000          1
SCOTT       3000          2
FORD        3000          2
JONES       2975          3
BLAKE       2850          4
CLARK       2450          5
ALLEN       1600          6
TURNER      1500          7
MILLER      1300          8
WARD        1250          9
MARTIN      1250          9
ADAMS       1100         10
JAMES        950         11
SMITH        800         12
```

As you can see from the previous result set, inline view X simply ranks the employees by SAL, while allowing for ties (because the solution uses DENSE_RANK instead of RANK, there are ties without gaps). The next step is to take the rows from inline view X and create groups by using a CASE expression to evaluate the ranking from DENSE_RANK. Additionally, use the window function ROW_NUMBER OVER to rank the employees by SAL within their group (within the group you are creating with the CASE expression). All of this happens in inline view Y and is shown here:

```
select ename,
       sal,
       rnk,
       case when rnk <= 3 then 1
            when rnk <= 6 then 2
            else                 3
       end grp,
       row_number()over (
         partition by case when rnk <= 3 then 1
                           when rnk <= 6 then 2
                           else                 3
                      end
            order by sal desc, ename
       ) grp_rnk
  from (
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp
       ) x
```

```
ENAME        SAL  RNK  GRP GRP_RNK
---------- ----- ---- ---- -------
KING        5000    1    1       1
FORD        3000    2    1       2
SCOTT       3000    2    1       3
JONES       2975    3    1       4
BLAKE       2850    4    2       1
CLARK       2450    5    2       2
ALLEN       1600    6    2       3
TURNER      1500    7    3       1
MILLER      1300    8    3       2
MARTIN      1250    9    3       3
WARD        1250    9    3       4
ADAMS       1100   10    3       5
JAMES        950   11    3       6
SMITH        800   12    3       7
```

Now the query is starting to take shape, and if you followed it from the beginning (from inline view X), you can see that it's not that complicated. The query so far returns each employee; their SAL; their RNK, which represents where their SAL ranks among all employees; their GRP, which indicates the group each employee is in (based on SAL); and finally GRP_RANK, which is a ranking (based on SAL) within their GRP.

At this point, perform a traditional pivot on ENAME while using the Oracle concatenation operator || to append the SAL. The function RPAD ensures that the numeric values in parentheses line up nicely. Finally, use GROUP BY on GRP_RNK to ensure you show each employee in the result set. The final result set is shown here:

```
select max(case grp when 1 then rpad(ename,6) ||
                    ' ('|| sal ||')' end) top_3,
       max(case grp when 2 then rpad(ename,6) ||
                    ' ('|| sal ||')' end) next_3,
       max(case grp when 3 then rpad(ename,6) ||
                    ' ('|| sal ||')' end) rest
  from (
select ename,
       sal,
       rnk,
       case when rnk <= 3 then 1
            when rnk <= 6 then 2
            else                  3
       end grp,
       row_number()over (
         partition by case when rnk <= 3 then 1
                           when rnk <= 6 then 2
                           else                 3
                      end
            Order by sal desc, ename
       ) grp_rnk
  from (
```

```
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp
       ) x
       ) y
group by grp_rnk

TOP_3           NEXT_3          REST
--------------- --------------- -------------
KING    (5000)  BLAKE   (2850)  TURNER (1500)
FORD    (3000)  CLARK   (2450)  MILLER (1300)
SCOTT   (3000)  ALLEN   (1600)  MARTIN (1250)
JONES   (2975)                  WARD   (1250)
ADAMS  (1100)
                                JAMES   (950)
                                SMITH   (800)
```

If you examine the queries in all of the steps, you'll notice that table EMP is accessed exactly once. One of the remarkable things about window functions is how much work you can do in just one pass through your data. There's no need for self-joins or temp tables; just get the rows you need and then let the window functions do the rest. Only in inline view X do you need to access EMP. From there, it's simply a matter of massaging the result set to look the way you want. Consider what all this means for performance if you can create this type of report with a single table access. Pretty cool.

14.9 Adding a Column Header into a Double Pivoted Result Set

Problem

You want to stack two result sets and then pivot them into two columns. Additionally, you want to add a "header" for each group of rows in each column. For example, you have two tables containing information about employees working in different areas of development in your company (say, in research and applications):

```
select * from it_research

DEPTNO ENAME
------ --------------------
   100 HOPKINS
   100 JONES
   100 TONEY
   200 MORALES
   200 P.WHITAKER
   200 MARCIANO
   200 ROBINSON
```

```
   300 LACY
   300 WRIGHT
   300 J.TAYLOR


select * from it_apps

DEPTNO ENAME
------ -----------------
   400 CORRALES
   400 MAYWEATHER
   400 CASTILLO
   400 MARQUEZ
   400 MOSLEY
   500 GATTI
   500 CALZAGHE
   600 LAMOTTA
   600 HAGLER
   600 HEARNS
   600 FRAZIER
   700 GUINN
   700 JUDAH
   700 MARGARITO
```

You would like to create a report listing the employees from each table in two columns. You want to return the DEPTNO followed by ENAME for each. Ultimately, you want to return the following result set:

```
RESEARCH             APPS
-------------------- ---------------
100                  400
  JONES                MAYWEATHER
  TONEY                CASTILLO
  HOPKINS              MARQUEZ
200                    MOSLEY
  P.WHITAKER           CORRALES
  MARCIANO           500
  ROBINSON             CALZAGHE
  MORALES              GATTI
300                  600
  WRIGHT               HAGLER
  J.TAYLOR             HEARNS
  LACY                 FRAZIER
                       LAMOTTA
                     700
                       JUDAH
                       MARGARITO
                       GUINN
```

Solution

For the most part, this solution requires nothing more than a simple stack 'n' pivot (union then pivot) with an added twist: the DEPTNO must precede the ENAME for each employee returned. The technique here uses a Cartesian product to generate an extra row for each DEPTNO, so you have the required rows necessary to show all employees, plus room for the DEPTNO. The solution uses Oracle syntax, but since DB2 supports window functions that can compute moving windows (the framing clause), converting this solution to work for DB2 is trivial. Because the IT_RESEARCH and IT_APPS tables exist only for this recipe, their table creation statements are shown along with this solution:

```
create table IT_research (deptno number, ename varchar2(20))

insert into IT_research values (100,'HOPKINS')
insert into IT_research values (100,'JONES')
insert into IT_research values (100,'TONEY')
insert into IT_research values (200,'MORALES')
insert into IT_research values (200,'P.WHITAKER')
insert into IT_research values (200,'MARCIANO')
insert into IT_research values (200,'ROBINSON')
insert into IT_research values (300,'LACY')
insert into IT_research values (300,'WRIGHT')
insert into IT_research values (300,'J.TAYLOR')


create table IT_apps (deptno number, ename varchar2(20))

insert into IT_apps values (400,'CORRALES')
insert into IT_apps values (400,'MAYWEATHER')
insert into IT_apps values (400,'CASTILLO')
insert into IT_apps values (400,'MARQUEZ')
insert into IT_apps values (400,'MOSLEY')
insert into IT_apps values (500,'GATTI')
insert into IT_apps values (500,'CALZAGHE')
insert into IT_apps values (600,'LAMOTTA')
insert into IT_apps values (600,'HAGLER')
insert into IT_apps values (600,'HEARNS')
insert into IT_apps values (600,'FRAZIER')
insert into IT_apps values (700,'GUINN')
insert into IT_apps values (700,'JUDAH')
insert into IT_apps values (700,'MARGARITO')
```

```
 1 select max(decode(flag2,0,it_dept)) research,
 2        max(decode(flag2,1,it_dept)) apps
 3   from (
 4 select sum(flag1)over(partition by flag2
 5                           order by flag1,rownum) flag,
 6        it_dept, flag2
 7   from (
```

```
 8 select 1 flag1, 0 flag2,
 9        decode(rn,1,to_char(deptno),' '||ename) it_dept
10   from (
11 select x.*, y.id,
12        row_number()over(partition by x.deptno order by y.id) rn
13   from (
14 select deptno,
15        ename,
16        count(*)over(partition by deptno) cnt
17   from it_research
18        ) x,
19        (select level id from dual connect by level <= 2) y
20        )
21  where rn <= cnt+1
22 union all
23 select 1 flag1, 1 flag2,
24        decode(rn,1,to_char(deptno),' '||ename) it_dept
25   from (
26 select x.*, y.id,
27        row_number()over(partition by x.deptno order by y.id) rn
28   from (
29 select deptno,
30        ename,
31        count(*)over(partition by deptno) cnt
32   from it_apps
33        ) x,
34        (select level id from dual connect by level <= 2) y
35        )
36  where rn <= cnt+1
37        ) tmp1
38        ) tmp2
39  group by flag
```

Discussion

Like many of the other warehousing/report type queries, the solution presented looks quite convoluted, but once broken down, you'll seen it's nothing more than a stack 'n' pivot with a Cartesian twist (on the rocks, with a little umbrella). The way to break down this query is to work on each part of the UNION ALL first and then bring it together for the pivot. Let's start with the lower portion of the UNION ALL:

```
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
```

```
       ) x,
       (select level id from dual connect by level <= 2) y
       ) z
 where rn <= cnt+1

FLAG1      FLAG2 IT_DEPT
----- ---------- -------------------------
    1          1 400
    1          1   MAYWEATHER
    1          1   CASTILLO
    1          1   MARQUEZ
    1          1   MOSLEY
    1          1   CORRALES
    1          1 500
    1          1   CALZAGHE
    1          1   GATTI
    1          1 600
    1          1   HAGLER
    1          1   HEARNS
    1          1   FRAZIER
    1          1   LAMOTTA
    1          1 700
    1          1   JUDAH
    1          1   MARGARITO
    1          1   GUINN
```

Let's examine exactly how that result set is put together. Breaking down the previous query to its simplest components, you have inline view X, which simply returns each ENAME and DEPTNO and the number of employees in each DEPTNO from table IT_APPS. The results are as follows:

```
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps

DEPTNO ENAME                       CNT
------ -------------------- ----------
   400 CORRALES                     5
   400 MAYWEATHER                   5
   400 CASTILLO                     5
   400 MARQUEZ                      5
   400 MOSLEY                       5
   500 GATTI                        2
   500 CALZAGHE                     2
   600 LAMOTTA                      4
   600 HAGLER                       4
   600 HEARNS                       4
   600 FRAZIER                      4
   700 GUINN                        3
   700 JUDAH                        3
   700 MARGARITO                    3
```

The next step is to create a Cartesian product between the rows returned from inline view X and two rows generated from DUAL using CONNECT BY. The results of this operation are as follows:

```
select *
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
 order by 2

DEPTNO ENAME      CNT  ID
------ ---------- --- ---
   500 CALZAGHE     2   1
   500 CALZAGHE     2   2
   400 CASTILLO     5   1
   400 CASTILLO     5   2
   400 CORRALES     5   1
   400 CORRALES     5   2
   600 FRAZIER      4   1
   600 FRAZIER      4   2
   500 GATTI        2   1
   500 GATTI        2   2
   700 GUINN        3   1
   700 GUINN        3   2
   600 HAGLER       4   1
   600 HAGLER       4   2
   600 HEARNS       4   1
   600 HEARNS       4   2
   700 JUDAH        3   1
   700 JUDAH        3   2
   600 LAMOTTA      4   1
   600 LAMOTTA      4   2
   700 MARGARITO    3   1
   700 MARGARITO    3   2
   400 MARQUEZ      5   1
   400 MARQUEZ      5   2
   400 MAYWEATHER   5   1
   400 MAYWEATHER   5   2
   400 MOSLEY       5   1
   400 MOSLEY       5   2
```

As you can see from these results, each row from inline view X is now returned twice due to the Cartesian product with inline view Y. The reason a Cartesian is needed will become clear shortly. The next step is to take the current result set and rank each employee within his DEPTNO by ID (ID has a value of 1 or 2 as was returned by the Cartesian product). The result of this ranking is shown in the output from the following query:

```
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y

DEPTNO ENAME      CNT ID          RN
------ ---------- --- --- ----------
   400 CORRALES     5   1          1
   400 MAYWEATHER   5   1          2
   400 CASTILLO     5   1          3
   400 MARQUEZ      5   1          4
   400 MOSLEY       5   1          5
   400 CORRALES     5   2          6
   400 MOSLEY       5   2          7
   400 MAYWEATHER   5   2          8
   400 CASTILLO     5   2          9
   400 MARQUEZ      5   2         10
   500 GATTI        2   1          1
   500 CALZAGHE     2   1          2
   500 GATTI        2   2          3
   500 CALZAGHE     2   2          4
   600 LAMOTTA      4   1          1
   600 HAGLER       4   1          2
   600 HEARNS       4   1          3
   600 FRAZIER      4   1          4
   600 LAMOTTA      4   2          5
   600 HAGLER       4   2          6
   600 FRAZIER      4   2          7
   600 HEARNS       4   2          8
   700 GUINN        3   1          1
   700 JUDAH        3   1          2
   700 MARGARITO    3   1          3
   700 GUINN        3   2          4
   700 JUDAH        3   2          5
   700 MARGARITO    3   2          6
```

Each employee is ranked; then his duplicate is ranked. The result set contains duplicates for all employees in table IT_APP, along with their ranking within their DEPTNO. The reason you need to generate these extra rows is because you need a slot in the result set to slip in the DEPTNO in the ENAME column. If you Cartesian-join IT_APPS with a one-row table, you get no extra rows (because cardinality of any table × 1 = cardinality of that table).

The next step is to take the results returned thus far and pivot the result set such that all the ENAMEs are returned in one column but are preceded by the DEPTNO they are in. The following query shows how this happens:

```
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       ) z
 where rn <= cnt+1

FLAG1      FLAG2 IT_DEPT
----- ---------- ------------------------
    1          1 400
    1          1   MAYWEATHER
    1          1   CASTILLO
    1          1   MARQUEZ
    1          1   MOSLEY
    1          1   CORRALES
    1          1 500
    1          1   CALZAGHE
    1          1   GATTI
    1          1 600
    1          1   HAGLER
    1          1   HEARNS
    1          1   FRAZIER
    1          1   LAMOTTA
    1          1 700
    1          1   JUDAH
    1          1   MARGARITO
    1          1   GUINN
```

FLAG1 and FLAG2 come into play later and can be ignored for the moment. Focus your attention on the rows in IT_DEPT. The number of rows returned for each DEPTNO is CNT*2, but all that is needed is CNT+1, which is the filter in the WHERE clause. RN is the ranking for each employee. The rows kept are all those ranked less than or equal to CNT+1; i.e., all employees in each DEPTNO plus one more (this extra employee is the employee who is ranked first in their DEPTNO). This extra row is where the DEPTNO will slide in. By using DECODE (an older Oracle function that gives more or less the equivalent of a CASE expression) to evaluate the value of RN, you can slide the value of DEPTNO into the result set. The employee who was at position one (based on the value of RN) is still shown in the result set, but is now last in each DEPTNO (because the order is irrelevant, this is not a problem). That pretty much covers the lower part of the UNION ALL.

The upper part of the UNION ALL is processed in the same way as the lower part, so there's no need to explain how that works. Instead, let's examine the result set returned when stacking the queries:

```
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1

FLAG1      FLAG2 IT_DEPT
----- ---------- -----------------------
    1          0 100
    1          0   JONES
    1          0   TONEY
    1          0   HOPKINS
    1          0 200
    1          0   P.WHITAKER
    1          0   MARCIANO
    1          0   ROBINSON
    1          0   MORALES
    1          0 300
    1          0   WRIGHT
    1          0   J.TAYLOR
    1          0   LACY
    1          1 400
    1          1   MAYWEATHER
    1          1   CASTILLO
```

```
     1          1   MARQUEZ
     1          1   MOSLEY
     1          1   CORRALES
     1          1 500
     1          1   CALZAGHE
     1          1   GATTI
     1          1 600
     1          1   HAGLER
     1          1   HEARNS
     1          1   FRAZIER
     1          1   LAMOTTA
     1          1 700
     1          1   JUDAH
     1          1   MARGARITO
     1          1   GUINN
```

At this point, it isn't clear what FLAG1's purpose is, but you can see that FLAG2 identifies which rows come from which part of the UNION ALL (0 for the upper part, 1 for the lower part).

The next step is to wrap the stacked result set in an inline view and create a running total on FLAG1 (finally, its purpose is revealed!), which will act as a ranking for each row in each stack. The results of the ranking (running total) are shown here:

```
select sum(flag1)over(partition by flag2
                      order by flag1,rownum) flag,
       it_dept, flag2
  from (
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
    from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
```

```
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
       ) tmp1


FLAG IT_DEPT              FLAG2
---- --------------- ----------
   1   100                    0
   2     JONES                0
   3     TONEY                0
   4     HOPKINS              0
   5   200                    0
   6     P.WHITAKER           0
   7     MARCIANO             0
   8     ROBINSON             0
   9     MORALES              0
  10    300                   0
  11     WRIGHT               0
  12     J.TAYLOR             0
  13     LACY                 0
   1   400                    1
   2     MAYWEATHER           1
   3     CASTILLO             1
   4     MARQUEZ              1
   5     MOSLEY               1
   6     CORRALES             1
   7   500                    1
   8     CALZAGHEe            1
   9     GATTI                1
  10   600                    1
  11     HAGLER               1
  12     HEARNS               1
  13     FRAZIER              1
  14     LAMOTTA              1
  15   700                    1
  16     JUDAH                1
  17     MARGARITO            1
  18     GUINN                1
```

The last step (finally!) is to pivot the value returned by TMP1 on FLAG2 while grouping by FLAG (the running total generated in TMP1). The results from TMP1 are wrapped in an inline view and pivoted (wrapped in a final inline view called TMP2). The ultimate solution and result set are shown here:

```
select max(decode(flag2,0,it_dept)) research,
       max(decode(flag2,1,it_dept)) apps
  from (
select sum(flag1)over(partition by flag2
                          order by flag1,rownum) flag,
```

```
       it_dept, flag2
  from (
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
       ) tmp1
       ) tmp2
 group by flag


RESEARCH             APPS
-------------------- ---------------
100                  400
  JONES                MAYWEATHER
  TONEY                CASTILLO
  HOPKINS              MARQUEZ
200                    MOSLEY
  P.WHITAKER           CORRALES
  MARCIANO           500
  ROBINSON             CALZAGHE
  MORALES              GATTI
300                  600
  WRIGHT               HAGLER
  J.TAYLOR             HEARNS
  LACY                 FRAZIER
                       LAMOTTA
```

```
700
  JUDAH
  MARGARITO
  GUINN
```

14.10 Converting a Scalar Subquery to a Composite Subquery in Oracle

Problem

You want to bypass the restriction of returning exactly one value from a scalar subquery. For example, you attempt to execute the following query:

```
select e.deptno,
       e.ename,
       e.sal,
       (select d.dname,d.loc,sysdate today
          from dept d
         where e.deptno=d.deptno)
  from emp e
```

but receive an error because subqueries in the SELECT list are allowed to return only a single value.

Solution

Admittedly, this problem is quite unrealistic, because a simple join between tables EMP and DEPT would allow you to return as many values you want from DEPT. Nevertheless, the key is to focus on the technique and understand how to apply it to a scenario that you find useful. The key to bypassing the requirement to return a single value when placing a SELECT within SELECT (scalar subquery) is to take advantage of Oracle's object types. You can define an object to have several attributes, and then you can work with it as a single entity or reference each element individually. In effect, you don't really bypass the rule at all. You simply return one value—an [.keep-together]#object—#that in turn contains many attributes.

This solution makes use of the following object type:

```
create type generic_obj
    as object (
    val1 varchar2(10),
    val2 varchar2(10),
    val3 date
 );
```

With this type in place, you can execute the following query:

```
1 select x.deptno,
2        x.ename,
```

```
 3        x.multival.val1 dname,
 4        x.multival.val2 loc,
 5        x.multival.val3 today
 6  from (
 7select e.deptno,
 8       e.ename,
 9       e.sal,
10       (select generic_obj(d.dname,d.loc,sysdate+1)
11          from dept d
12         where e.deptno=d.deptno) multival
13  from emp e
14       ) x

DEPTNO ENAME      DNAME      LOC        TODAY
------ ---------- ---------- ---------- -----------
    20 SMITH      RESEARCH   DALLAS     12-SEP-2020
    30 ALLEN      SALES      CHICAGO    12-SEP-2020
    30 WARD       SALES      CHICAGO    12-SEP-2020
    20 JONES      RESEARCH   DALLAS     12-SEP-2020
    30 MARTIN     SALES      CHICAGO    12-SEP-2020
    30 BLAKE      SALES      CHICAGO    12-SEP-2020
    10 CLARK      ACCOUNTING NEW YORK   12-SEP-2020
    20 SCOTT      RESEARCH   DALLAS     12-SEP-2020
    10 KING       ACCOUNTING NEW YORK   12-SEP-2020
    30 TURNER     SALES      CHICAGO    12-SEP-2020
    20 ADAMS      RESEARCH   DALLAS     12-SEP-2020
    30 JAMES      SALES      CHICAGO    12-SEP-2020
    20 FORD       RESEARCH   DALLAS     12-SEP-2020
    10 MILLER     ACCOUNTING NEW YORK   12-SEP-2020
```

Discussion

The key to the solution is to use the object's constructor function (by default the constructor function has the same name as the object). Because the object itself is a single scalar value, it does not violate the scalar subquery rule, as you can see from the following:

```
select e.deptno,
       e.ename,
       e.sal,
       (select generic_obj(d.dname,d.loc,sysdate-1)
          from dept d
       where e.deptno=d.deptno) multival
from emp e

DEPTNO ENAME  SAL   MULTIVAL(VAL1, VAL2, VAL3)
------ ------ ----- -------------------------------------------------------
    20 SMITH   800  GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
    30 ALLEN  1600  GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
    30 WARD   1250  GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
    20 JONES  2975  GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
    30 MARTIN 1250  GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
```

```
30 BLAKE  2850  GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
10 CLARK  2450  GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2020')
20 SCOTT  3000  GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
10 KING   5000  GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2020')
30 TURNER 1500  GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
20 ADAMS  1100  GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
30 JAMES   950  GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
20 FORD   3000  GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
10 MILLER 1300  GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2020')
```

The next step is to simply wrap the query in an inline view and extract the attributes.

In Oracle, unlike the case with other vendors, you do not generally need to name your inline views. In this particular case, however, you do need to name your inline view. Otherwise, you will not be able to reference the object's attributes.

14.11 Parsing Serialized Data into Rows

Problem

You have serialized data (stored in strings) that you want to parse and return as rows. For example, you store the following data:

```
STRINGS
-----------------------------------
entry:stewiegriffin:lois:brian:
entry:moe::sizlack:
entry:petergriffin:meg:chris:
entry:willie:
entry:quagmire:mayorwest:cleveland:
entry:::flanders:
entry:robo:tchi:ken:
```

You want to convert these serialized strings into the following result set:

```
VAL1            VAL2            VAL3
--------------- --------------- ---------------
moe                             sizlack
petergriffin    meg             chris
quagmire        mayorwest       cleveland
robo            tchi            ken
stewiegriffin   lois            brian
willie
                                flanders
```

Solution

Each serialized string in this example can store up to three values. The values are delimited by colons, and a string may or may not have all three entries. If a string does not have all three entries, you must be careful to place the entries that are available into the correct column in the result set. For example, consider the following row:

```
entry:::flanders:
```

This row represents an entry with the first two values missing and only the third value available. Hence, if you examine the target result set in the "Problem" section, you will notice that for the row FLANDERS is in, both VAL1 and VAL2 are NULL.

The key to this solution is nothing more than a string walk with some string parsing, following by a simple pivot. This solution uses rows from view V, which is defined as follows. The example uses Oracle syntax, but since nothing more than string parsing functions are needed for this recipe, converting to other platforms is simple:

```
create view V
    as
select 'entry:stewiegriffin:lois:brian:' strings
  from dual
 union all
select 'entry:moe::sizlack:'
  from dual
 union all
select 'entry:petergriffin:meg:chris:'
  from dual
 union all
select 'entry:willie:'
  from dual
 union all
select 'entry:quagmire:mayorwest:cleveland:'
  from dual
 union all
select 'entry:::flanders:'
  from dual
 union all
select 'entry:robo:tchi:ken:'
  from dual
```

Using view V to supply the example data to parse, the solution is as follows:

```
 1  with cartesian as (
 2  select level id
 3    from dual
 4   connect by level <= 100
 5  )
 6  select max(decode(id,1,substr(strings,p1+1,p2-1))) val1,
 7         max(decode(id,2,substr(strings,p1+1,p2-1))) val2,
 8         max(decode(id,3,substr(strings,p1+1,p2-1))) val3
```

```
 9   from (
10 select v.strings,
11        c.id,
12        instr(v.strings,':',1,c.id) p1,
13        instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
14   from v, cartesian c
15  where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
16        )
17  group by strings
18  order by 1
```

Discussion

The first step is to walk the serialized strings:

```
with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select v.strings,
     c.id
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1

STRINGS                            ID
---------------------------------- ---
entry:::flanders:                   1
entry:::flanders:                   2
entry:::flanders:                   3
entry:moe::sizlack:                 1
entry:moe::sizlack:                 2
entry:moe::sizlack:                 3
entry:petergriffin:meg:chris:       1
entry:petergriffin:meg:chris:       3
entry:petergriffin:meg:chris:       2
entry:quagmire:mayorwest:cleveland: 1
entry:quagmire:mayorwest:cleveland: 3
entry:quagmire:mayorwest:cleveland: 2
entry:robo:tchi:ken:                1
entry:robo:tchi:ken:                2
entry:robo:tchi:ken:                3
entry:stewiegriffin:lois:brian:     1
entry:stewiegriffin:lois:brian:     3
entry:stewiegriffin:lois:brian:     2
entry:willie:                       1
```

The next step is to use the function INSTR to find the numeric position of each colon in each string. Since each value you need to extract is enclosed by two colons, the numeric values are aliased P1 and P2, for “position one” and “position two”:

```
with cartesian as (
select level id
  from dual
  connect by level <= 100
)
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
       from v,cartesian c
      where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
      order by 1

STRINGS                            ID          P1         P2
---------------------------------- --- ---------- ----------
entry:::flanders:                    1          6          1
entry:::flanders:                    2          7          1
entry:::flanders:                    3          8          9
entry:moe::sizlack:                  1          6          4
entry:moe::sizlack:                  2         10          1
entry:moe::sizlack:                  3         11          8
entry:petergriffin:meg:chris:        1          6         13
entry:petergriffin:meg:chris:        3         23          6
entry:petergriffin:meg:chris:        2         19          4
entry:quagmire:mayorwest:cleveland:  1          6          9
entry:quagmire:mayorwest:cleveland:  3         25         10
entry:quagmire:mayorwest:cleveland:  2         15         10
entry:robo:tchi:ken:                 1          6          5
entry:robo:tchi:ken:                 2         11          5
entry:robo:tchi:ken:                 3         16          4
entry:stewiegriffin:lois:brian:      1          6         14
entry:stewiegriffin:lois:brian:      3         25          6
entry:stewiegriffin:lois:brian:      2         20          5
entry:willie:                        1          6          7
```

Now that you know the numeric positions for each pair of colons in each string, simply pass the information to the function SUBSTR to extract values. Since you want to create a result set with three columns, use DECODE to evaluate the ID from the Cartesian product:

```
with cartesian as (
  select level id
    from dual
   connect by level <= 100
  )
  select decode(id,1,substr(strings,p1+1,p2-1)) val1,
         decode(id,2,substr(strings,p1+1,p2-1)) val2,
         decode(id,3,substr(strings,p1+1,p2-1)) val3
    from (
  select v.strings,
         c.id,
         instr(v.strings,':',1,c.id) p1,
```

```
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
       )
 order by 1

VAL1            VAL2            VAL3
--------------- --------------- --------------
moe
petergriffin
quagmire
robo
stewieggriffin
willie
                lois

                meg
                mayorwest

                tchi
                                brian
                                sizlack
                                chris
                                cleveland
                                flanders
                                ken
```

The last step is to apply an aggregate function to the values returned by SUBSTR while grouping by ID, to make a human-readable result set:

```
with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select max(decode(id,1,substr(strings,p1+1,p2-1))) val1,
       max(decode(id,2,substr(strings,p1+1,p2-1))) val2,
       max(decode(id,3,substr(strings,p1+1,p2-1))) val3
  from (
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
 from v,cartesian c
where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
       )
group by strings
order by 1

VAL1            VAL2            VAL3
--------------- --------------- -----------
moe                             sizlack
petergriffin    meg             chris
```

```
quagmire        mayorwest       cleveland
robo            tchi            ken
stewiegriffin   lois            brian
willie
                                flanders
```

14.12 Calculating Percent Relative to Total

Problem

You want to report a set of numeric values, and you want to show each value as a percentage of the whole. For example, you are on an Oracle system and you want to return a result set that shows the breakdown of salaries by JOB so that you can determine which JOB position costs the company the most money. You also want to include the number of employees per JOB to prevent the results from being misleading. You want to produce the following report:

```
JOB         NUM_EMPS PCT_OF_ALL_SALARIES
--------- ---------- -------------------
CLERK              4                  14
ANALYST            2                  20
MANAGER            3                  28
SALESMAN           4                  19
PRESIDENT          1                  17
```

As you can see, if the number of employees is not included in the report, it looks as if the president position takes very little of the overall salary. Seeing that there is only one president helps put into perspective what that 17% means.

Solution

Only Oracle enables a decent solution to this problem, which involves using the built-in function RATIO_TO_REPORT. To calculate percentages of the whole for other databases, you can use division as shown in Recipe 7.11:

```
select job,num_emps,sum(round(pct)) pct_of_all_salaries
 from (
select job,
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp
       )
 group by job,num_emps
```

Discussion

The first step is to use the window function COUNT OVER to return the number of employees per JOB. Then use RATIO_TO_REPORT to return the percentage each salary counts against the total (the value is returned in decimal):

```
select job,
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp

JOB         NUM_EMPS        PCT
--------- ---------- ----------
ANALYST            2 10.3359173
ANALYST            2 10.3359173
CLERK              4 2.75624462
CLERK              4 3.78983635
CLERK              4  4.4788975
CLERK              4 3.27304048
MANAGER            3 10.2497847
MANAGER            3 8.44099914
MANAGER            3 9.81912145
PRESIDENT          1 17.2265289
SALESMAN           4 5.51248923
SALESMAN           4 4.30663221
SALESMAN           4 5.16795866
SALESMAN           4 4.30663221
```

The last step is to use the aggregate function SUM to sum the values returned by RATIO_TO_REPORT. Be sure to group by JOB and NUM_EMPS. Multiply by 100 to return a whole number that represents a percentage (e.g., to return 25 rather than 0.25 for 25%):

```
select job,num_emps,sum(round(pct)) pct_of_all_salaries
  from (
select job,
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp
       )
 group by job,num_emps

JOB         NUM_EMPS PCT_OF_ALL_SALARIES
--------- ---------- -------------------
CLERK              4                  14
ANALYST            2                  20
MANAGER            3                  28
SALESMAN           4                  19
PRESIDENT          1                  17
```

14.13 Testing for Existence of a Value Within a Group

Problem

You want to create a Boolean flag for a row depending on whether any row in its group contains a specific value. Consider an example of a student who has taken a certain number of exams during a period of time. A student will take three exams over three months. If a student passes one of these exams, the requirement is satisfied and a flag should be returned to express that fact. If a student did not pass any of the three tests in the three-month period, then an additional flag should be returned to express that fact as well. Consider the following example (using Oracle syntax to make up rows for this example; minor modifications are necessary for the other vendors, making user of window functions):

```
create view V
as
select 1 student_id,
       1 test_id,
       2 grade_id,
       1 period_id,
       to_date('02/01/2020','MM/DD/YYYY') test_date,
       0 pass_fail
  from dual union all
select 1, 2, 2, 1, to_date('03/01/2020','MM/DD/YYYY'), 1 from dual union all
select 1, 3, 2, 1, to_date('04/01/2020','MM/DD/YYYY'), 0 from dual union all
select 1, 4, 2, 2, to_date('05/01/2020','MM/DD/YYYY'), 0 from dual union all
select 1, 5, 2, 2, to_date('06/01/2020','MM/DD/YYYY'), 0 from dual union all
select 1, 6, 2, 2, to_date('07/01/2020','MM/DD/YYYY'), 0 from dual

select *
  from V

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   PASS_FAIL
---------- ------- -------- --------- ----------- ---------
         1       1        2         1 01-FEB-2020         0
         1       2        2         1 01-MAR-2020         1
         1       3        2         1 01-APR-2020         0
         1       4        2         2 01-MAY-2020         0
         1       5        2         2 01-JUN-2020         0
         1       6        2         2 01-JUL-2020         0
```

Examining the previous result set, you see that the student has taken six tests over two, three-month periods. The student has passed one test (1 means "pass"; 0 means "fail"); thus, the requirement is satisfied for the entire first period. Because the student did not pass any exams during the second period (the next three months), PASS_FAIL is 0 for all three exams. You want to return a result set that highlights whether a student has passed a test for a given period. Ultimately you want to return the following result set:

```
STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   METREQ IN_PROGRESS
---------- ------- -------- --------- ----------- ------ -----------
         1       1        2         1 01-FEB-2020      +           0
         1       2        2         1 01-MAR-2020      +           0
         1       3        2         1 01-APR-2020      +           0
         1       4        2         2 01-MAY-2020      -           0
         1       5        2         2 01-JUN-2020      -           0
         1       6        2         2 01-JUL-2020      -           1
```

The values for METREQ ("met requirement") are + and –, signifying the student either has or has not satisfied the requirement of passing at least one test in a period (three-month span), respectively. The value for IN_PROGRESS should be 0 if a student has already passed a test in a given period. If a student has not passed a test for a given period, then the row that has the latest exam date for that student will have a value of 1 for IN_PROGRESS.

Solution

This problem appears tricky because you have to treat rows in a group as a group and not as individuals. Consider the values for PASS_FAIL in the "Problem" section. If you evaluate row by row, it appears that the value for METREQ for each row except TEST_ID 2 should be –, when it's not the case. You must ensure you evaluate the rows as a group. By using the window function MAX OVER, you can easily determine whether a student passed at least one test during a particular period. Once you have that information, the "Boolean" values are a simple matter of using CASE expressions:

```
 1  select student_id,
 2         test_id,
 3         grade_id,
 4         period_id,
 5         test_date,
 6         decode( grp_p_f,1,lpad('+',6),lpad('-',6) ) metreq,
 7         decode( grp_p_f,1,0,
 8                 decode( test_date,last_test,1,0 ) ) in_progress
 9   from (
10 select V.*,
11        max(pass_fail)over(partition by
12                      student_id,grade_id,period_id) grp_p_f,
13        max(test_date)over(partition by
14                      student_id,grade_id,period_id) last_test
15   from V
16        ) x
```

Discussion

The key to the solution is using the window function MAX OVER to return the greatest value of PASS_FAIL for each group. Because the values for PASS_FAIL are

only 1 or 0, if a student passed at least one exam, then MAX OVER would return 1 for the entire group. How this works is shown here:

```
select V.*,
       max(pass_fail)over(partition by
                       student_id,grade_id,period_id) grp_pass_fail
  from V

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   PASS_FAIL GRP_PASS_FAIL
---------- ------- -------- --------- ----------- --------- -------------
         1       1        2         1 01-FEB-2020         0             1
         1       2        2         1 01-MAR-2020         1             1
         1       3        2         1 01-APR-2020         0             1
         1       4        2         2 01-MAY-2020         0             0
         1       5        2         2 01-JUN-2020         0             0
         1       6        2         2 01-JUL-2020         0             0
```

The previous result set shows that the student passed at least one test during the first period; thus, the entire group has a value of 1 or "pass." The next requirement is that if the student has not passed any tests in a period, return a value of 1 for the IN_PROGRESS flag for the latest test date in that group. You can use the window function MAX OVER to do this as well:

```
select V.*,
       max(pass_fail)over(partition by
                       student_id,grade_id,period_id) grp_p_f,
       max(test_date)over(partition by
                       student_id,grade_id,period_id) last_test
  from V

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   PASS_FAIL GRP_P_F LAST_TEST
---------- ------- -------- --------- ----------- --------- ------- -----------
         1       1        2         1 01-FEB-2020         0       1 01-APR-2020
         1       2        2         1 01-MAR-2020         1       1 01-APR-2020
         1       3        2         1 01-APR-2020         0       1 01-APR-2020
         1       4        2         2 01-MAY-2020         0       0 01-JUL-2020
         1       5        2         2 01-JUN-2020         0       0 01-JUL-2020
         1       6        2         2 01-JUL-2020         0       0 01-JUL-2020
```

Now that you have determined for which period the student has passed a test and what the latest test date for each period is, the last step is simply a matter of applying some formatting magic to make the result set look nice. The ultimate solution uses Oracle's DECODE function (CASE supporters, eat your hearts out) to create the METREQ and IN_PROGRESS columns. Use the LPAD function to right justify the values for METREQ:

```
select student_id,
       test_id,
       grade_id,
       period_id,
       test_date,
```

```
       decode( grp_p_f,1,lpad('+',6),lpad('-',6) ) metreq,
       decode( grp_p_f,1,0,
               decode( test_date,last_test,1,0 ) ) in_progress
  from (
select V.*,
       max(pass_fail)over(partition by
                       student_id,grade_id,period_id) grp_p_f,
       max(test_date)over(partition by
                       student_id,grade_id,period_id) last_test
  from V
       ) x

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   METREQ IN_PROGRESS
---------- ------- -------- --------- ----------- ------ -----------
         1       1        2         1 01-FEB-2020      +           0
         1       2        2         1 01-MAR-2020      +           0
         1       3        2         1 01-APR-2020      +           0
         1       4        2         2 01-MAY-2020      -           0
         1       5        2         2 01-JUN-2020      -           0
         1       6        2         2 01-JUL-2020      -           1
```

14.14 Summing Up

SQL is more powerful than many credit it. Throughout this book we have tried to challenge you to see more applications than are typically noted. In this chapter, we've headed straight for the edge cases and tried to show just how you can push SQL, both with standard features and with certain vendor-specific features.

APPENDIX A

Window Function Refresher

The recipes in this book take full advantage of the window functions added to the ISO SQL standard in 2003, as well as vendor-specific window functions. This appendix is meant to serve as a brief overview of how window functions work. Window functions make many typically difficult tasks (difficult to solve using standard SQL, that is) quite easy. For a complete list of window functions available, full syntax, and in-depth coverage of how they work, please consult your vendor's documentation.

Grouping

Before moving on to window functions, it is crucial that you understand how grouping works in SQL—the concept of grouping results in SQL can be difficult to master. The problems stem from not fully understanding how the GROUP BY clause works and why certain queries return certain results when using GROUP BY.

Simply stated, grouping is a way to organize like rows together. When you use GROUP BY in a query, each row in the result set is a group and represents one or more rows with the same values in one or more columns that you specify. That's the gist of it.

If a group is simply a unique instance of a row that represents one or more rows with the same value for a particular column (or columns), then practical examples of groups from table EMP include *all employees in department 10* (the common value for these employees that enables them to be in the same group is DEPTNO=10) or *all clerks* (the common value for these employees that enables them to be in the same group is JOB=*CLERK*). Consider the following queries. The first shows all employees in department 10; the second query groups the employees in department 10 and returns the following information about the group: the number of rows (members) in the group, the highest salary, and the lowest salary:

```
select deptno,ename
  from emp
 where deptno=10

DEPTNO ENAME
------ ----------
    10 CLARK
    10 KING
    10 MILLER


select deptno,
       count(*) as cnt,
       max(sal) as hi_sal,
       min(sal) as lo_sal
  from emp
 where deptno=10
 group by deptno

DEPTNO        CNT     HI_SAL     LO_SAL
------ ---------- ---------- ----------
    10          3       5000       1300
```

If you were not able to group the employees in department 10 together, to get the information in the second query, you would have to manually inspect the rows for that department (trivial if there are only three rows, but what if there were three million rows?). So, why would anyone want to group? Reasons for doing so vary; perhaps you want to see how many different groups exist or how many members (rows) are in each group. As you can see from this simple example, grouping allows you to get information about many rows in a table without having to inspect them one by one.

Definition of an SQL Group

In mathematics, a group is defined, for the most part, as (G, •, e), where G is a set, • is a binary operation in G, and e is a member of G. We will use this definition as the foundation for what a SQL group is. A SQL group will be defined as (G, e), where G is a result set of a single or self-contained query that uses GROUP BY, e is a member of G, and the following axioms are satisfied:

- For each e in G, e is distinct and represents one or more instances of e.
- For each e in G, the aggregate function COUNT returns a value > 0.

The result set is included in the definition of a SQL group to reinforce the fact that we are defining what groups are when working with queries only. Thus, it would be accurate to replace e in each axiom with the word *row* because the rows in the result set are technically the groups.

Because these properties are fundamental to what we consider a group, it is important that we prove they are true (and we will proceed to do so through the use of some example SQL queries).

Groups are nonempty

By its very definition, a group must have at least one member (or row). If we accept this as a truth, then it can be said that a group cannot be created from an empty table. To prove that proposition true, simply try to prove it is false. The following example creates an empty table and then attempts to create groups via three different queries against that empty table:

```
create table fruits (name varchar(10))

select name
  from fruits
 group by name

(no rows selected)


select count(*) as cnt
  from fruits
 group by name

(no rows selected)


select name, count(*) as cnt
  from fruits
 group by name

(no rows selected)
```

As you can see from these queries, it is impossible to create what SQL considers a group from an empty table.

Groups are distinct

Now let's prove that the groups created via queries with a GROUP BY clause are distinct. The following example inserts five rows into table FRUITS and then creates groups from those rows:

```
insert into fruits values ('Oranges')
insert into fruits values ('Oranges')
insert into fruits values ('Oranges')
insert into fruits values ('Apple')
insert into fruits values ('Peach')

select *
  from fruits
```

```
NAME
--------
Oranges
Oranges
Oranges
Apple
Peach

select name
  from fruits
 group by name

NAME
-------
Apple
Oranges
Peach

select name, count(*) as cnt
  from fruits
 group by name

NAME       CNT
------- --------
Apple          1
Oranges        3
Peach          1
```

The first query shows that "Oranges" occurs three times in table FRUITS. However, the second and third queries (using GROUP BY) return only one instance of "Oranges." Taken together, these queries prove that the rows in the result set (*e* in *G*, from our definition) are distinct, and each value of NAME represents one or more instances of itself in table FRUITS.

Knowing that groups are distinct is important because it means, typically, you would not use the DISTINCT keyword in your SELECT list when using a GROUP BY in your queries.

We don't pretend GROUP BY and DISTINCT are the same. They represent two completely different concepts. We do state that the items listed in the GROUP BY clause will be distinct in the result set and that using DISTINCT as well as GROUP BY is redundant.

Frege's Axiom and Russell's Paradox

For those of you who are interested, Frege's *axiom of abstraction*, based on Cantor's solution for defining set membership for infinite or uncountable sets, states that, given a specific identifying property, there exists a set whose members are only those items having that property. The source of trouble, as put by Robert Stoll, "is the unrestricted use of the principal of abstraction." Bertrand Russell asked Gottlob Frege to consider a set whose members are sets and have the defining property of not being members of themselves.

As Russell pointed out, the axiom of abstraction gives too much freedom because you are simply specifiying a condition or property to define set membership; thus, a contradiction can be found. To better explain how a contradiction can be found, he devised the "Barber puzzle." The Barber puzzle states:

> In a certain town there is a male barber who shaves all those men, and only those men, who do not shave themselves. If this is true, who, then, shaves the barber?

For a more concrete example, consider the set that can be described as:

> For all members *x* in *y* that satisfy a specific condition (*P*).

The mathematical notation for this description is:

```
{x e y | P(x)}
```

Because the previous set considers *only those x in y that satisfy a condition (P)*, you may find it more intuitive to describe the set as *x is a member of y if and only if x satisfies a condition (P)*.

At this point let us define this condition *P(x)* as *x is not a member of x*:

```
( x e x )
```

The set is now defined as *x is a member of y if and only if x is not a member of x*:

```
{x e y | ( x e x )}
```

Russell's paradox may not be clear to you yet, but ask yourself this: can the previous set be a member of itself? Let's assume that $x = y$ and look at the set again. The following set can be defined as *y is a member of y if and only if y is not a member of y*:

```
{y e y | ( y e y )}
```

Simply put, Russell's paradox leaves us in a position to have a set that is concurrently a member of itself and not a member of itself, which is a contradiction. Intuitive thinking would lead one to believe this isn't a problem at all; indeed, how can a set be a member of itself? The set of all books, after all, is not a book. So why does this paradox exist, and how can it be an issue? It becomes an issue when you consider more abstract applications of set theory. For example, a "practical" application of Russell's paradox can be demonstrated by considering the set of all sets. If we allow such a

concept to exist, then by its very definition, it must be a member of itself (it is, after all, the set of all sets). What then happens when you apply the previous *P(x)* to the set of all sets? Simply stated, Russell's paradox would state that the set of all sets is a member of itself if and only if it is not a member of itself—clearly a contradiction.

For those of you who are interested, Ernst Zermelo developed the axiom schema of separation (also referred to as the *axiom schema of subsets* or the *axiom of specification*) to elegantly sidestep Russell's paradox in axiomatic set theory.

COUNT is never zero

The queries and results in the preceding section also prove the final axiom that the aggregate function COUNT will never return zero when used in a query with GROUP BY on a nonempty table. It should not be surprising that you cannot return a count of zero for a group. We have already proved that a group cannot be created from an empty table; thus, a group must have at least one row. If at least one row exists, then the count will always be at least one.

Remember, we are talking about using COUNT with GROUP BY, not COUNT by itself. A query using COUNT without a GROUP BY on an empty table will, of course, return zero.

Paradoxes

The following quote is from Gottlob Frege in response to Bertrand Russell's discovery of a contradiction to Frege's axiom of abstraction in set theory:

> Hardly anything more unfortunate can befall a scientific writer than to have one of the foundations of his edifice shaken after the work is finished…. This was the position I was placed in by a letter of Mr. Bertrand Russell, just when the printing of this volume was nearing its completion.

Paradoxes many times provide scenarios that would seem to contradict established theories or ideas. In many cases these contradictions are localized and can be "worked around," or they are applicable to such small test cases that they can be safely ignored.

You may have guessed by now that the point to all this discussion of paradoxes is that there exists a paradox concerning our definition of an SQL group, and that paradox must be addressed. Although our focus right now is on groups, ultimately we are discussing SQL queries. In its GROUP BY clause, a query may have a wide range of values such as constants, expressions, or, most commonly, columns from a table. We pay a price for this flexibility, because NULL is a valid "value" in SQL. NULLs present problems because they are effectively ignored by aggregate functions. With that said,

if a table consists of a single row and its value is NULL, what would the aggregate function COUNT return when used in a GROUP BY query? By our very definition, when using GROUP BY and the aggregate function COUNT, a value >= 1 must be returned. What happens, then, in the case of values ignored by functions such as COUNT, and what does this mean to our definition of a GROUP? Consider the following example, which reveals the NULL group paradox (using the function COALESCE when necessary for readability):

```
select *
  from fruits

NAME
-------
Oranges
Oranges
Oranges
Apple
Peach

insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)

select coalesce(name,'NULL') as name
  from fruits

NAME
--------
Oranges
Oranges
Oranges
Apple
Peach
NULL
NULL
NULL
NULL
NULL

select coalesce(name,'NULL') as name,
       count(name) as cnt
  from fruits
 group by name

NAME            CNT
-------- ----------
Apple             1
NULL              0
```

```
Oranges          3
Peach            1
```

It would seem that the presence of NULL values in our table introduces a contradiction, or paradox, to our definition of a SQL group. Fortunately, this contradiction is not a real cause for concern, because the paradox has more to do with the implementation of aggregate functions than our definition. Consider the final query in the preceding set; a general problem statement for that query would be:

> Count the number of times each name occurs in table FRUITS or count the number of members in each group.

Examining the previous INSERT statements, it's clear that there are five rows with NULL values, which means there exists a NULL group with five members.

While NULL certainly has properties that differentiate it from other values, it is nevertheless a value and can in fact be a group.

How, then, can we write the query to return a count of 5 instead of 0, thus returning the information we are looking for while conforming to our definition of a group? The following example shows a workaround to deal with the NULL group paradox:

```
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name

NAME           CNT
--------- --------
Apple            1
Oranges          3
Peach            1
NULL             5
```

The workaround is to use COUNT(*) rather than COUNT(NAME) to avoid the NULL group paradox. Aggregate functions will ignore NULL values if any exist in the column passed to them. Thus, to avoid a zero when using COUNT, do not pass the column name; instead, pass in an asterisk (*). The * causes the COUNT function to count rows rather than the actual column values, so whether the actual values are NULL or not NULL is irrelevant.

One more paradox has to do with the axiom that each group in a result set (for each *e* in *G*) is distinct. Because of the nature of SQL result sets and tables, which are more accurately defined as multisets or "bags," not sets (because duplicate rows are allowed), it is possible to return a result set with duplicate groups. Consider the following queries:

```
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name
 union all
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name

NAME       CNT
---------- ---------
Apple            1
Oranges          3
Peach            1
NULL             5
Apple            1
Oranges          3
Peach            1
NULL             5


select x.*
  from (
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name
       ) x,
       (select deptno from dept) y

NAME       CNT
---------- ----------
Apple             1
Apple             1
Apple             1
Apple             1
Oranges           3
Oranges           3
Oranges           3
Oranges           3
Peach             1
Peach             1
Peach             1
Peach             1
NULL              5
NULL              5
NULL              5
NULL              5
```

As you can see in these queries, the groups are in fact repeated in the final results. Fortunately, this is not much to worry about because it represents only a partial

paradox. The first property of a group states that for (*G*, *e*), *G* is a result set from a single or self-contained query that uses GROUP BY. Simply put, the result set from any GROUP BY query itself conforms to our definition of a group. It is only when you combine the result sets from two GROUP BY queries to create a multiset that groups may repeat. The first query in the preceding example uses UNION ALL, which is not a set operation but a multiset operation, and invokes GROUP BY twice, effectively executing two queries.

If you use UNION, which is a set operation, you will not see repeating groups.

The second query in the preceding set uses a Cartesian product, which only works if you materialize the group first and then perform the Cartesian. Thus, the GROUP BY query when self-contained conforms to our definition. Neither of the two examples takes anything away from the definition of a SQL group. They are shown for completeness, and so that you can be aware that almost anything is possible in SQL.

Relationship Between SELECT and GROUP BY

With the concept of a group defined and proved, it is now time to move on to more practical matters concerning queries using GROUP BY. It is important to understand the relationship between the SELECT clause and the GROUP BY clause when grouping in SQL. It is important to keep in mind when using aggregate functions such as COUNT that any item in your SELECT list that is not used as an argument to an aggregate function must be part of your group. For example, if you write a SELECT clause such as this:

```
select deptno, count(*) as cnt
  from emp
```

then you must list DEPTNO in your GROUP BY clause:

```
select deptno, count(*) as cnt
  from emp
 group by deptno

DEPTNO  CNT
------- ----
     10    3
     20    5
     30    6
```

Constants, scalar values returned by user-defined functions, window functions, and noncorrelated scalar subqueries are exceptions to this rule. Since the SELECT clause

is evaluated after the GROUP BY clause, these constructs are allowed in the SELECT list and do not have to (and in some cases cannot) be specified in the GROUP BY clause. For example:

```
select 'hello' as msg,
       1 as num,
       deptno,
       (select count(*) from emp) as total,
       count(*) as cnt
  from emp
 group by deptno

MSG        NUM DEPTNO TOTAL CNT
-----  --- ------ ----- ---
hello      1     10    14   3
hello      1     20    14   5
hello      1     30    14   6
```

Don't let this query confuse you. The items in the SELECT list not listed in the GROUP BY clause do not change the value of CNT for each DEPTNO, nor do the values for DEPTNO change. Based on the results of the preceding query, we can define the rule about matching items in the SELECT list and the GROUP BY clause when using aggregates a bit more precisely:

> Items in a SELECT list that can potentially change the group or change the value returned by an aggregate function must be included in the GROUP BY clause.

The additional items in the preceding SELECT list did not change the value of CNT for any group (each DEPTNO), nor did they change the groups themselves.

Now it's fair to ask: exactly what items in a SELECT list can change a grouping or the value returned by an aggregate function? The answer is simple: other columns from the table(s) you are selecting from. Consider the prospect of adding the JOB column to the query we've been looking at:

```
select deptno, job, count(*) as cnt
  from emp
 group by deptno, job

DEPTNO JOB        CNT
------ ---------- ----
    10 CLERK         1
    10 MANAGER       1
    10 PRESIDENT     1
    20 CLERK         2
    20 ANALYST       2
    20 MANAGER       1
    30 CLERK         1
    30 MANAGER       1
    30 SALESMAN      4
```

By listing another column, JOB, from table EMP, we are changing the group and changing the result set. Thus, we must now include JOB in the GROUP BY clause along with DEPTNO; otherwise, the query will fail. The inclusion of JOB in the SELECT/GROUP BY clauses changes the query from "How many employees are in each department?" to "How many different types of employees are in each department?" Notice again that the groups are distinct; the values for DEPTNO and JOB *individually* are not distinct, but the combination of the two (which is what is in the GROUP BY and SELECT list, and thus in the group) is distinct (e.g., 10 and CLERK appear only once).

If you choose not to put items other than aggregate functions in the SELECT list, then you may list any valid column you want in the GROUP BY clause. Consider the following two queries, which highlight this fact:

```
select count(*)
  from emp
 group by deptno

 COUNT(*)
---------
        3
        5
        6

select count(*)
  from emp
 group by deptno,job

  COUNT(*)
----------
         1
         1
         1
         2
         2
         1
         1
         1
         4
```

Including items other than aggregate functions in the SELECT list is not mandatory, but often improves readability and usability of the results.

As a rule, when using GROUP BY and aggregate functions, any items in the SELECT list (from the table(s) in the FROM clause) not used as an argument to an aggregate function must be included in the GROUP BY clause. However, MySQL has a "feature" that allows you to deviate from this rule, allowing you to place items in your SELECT list (that are columns in the table(s) you are selecting from) that are not used as arguments to an aggregate function and that are not present in your GROUP BY clause. We use the term *feature* loosely here as its use is a bug waiting to happen. As a matter of fact, if you use MySQL and care at all about the accuracy of your queries, we suggest you urge them to remove this, ahem, "feature."

Windowing

Once you understand the concept of grouping and using aggregates in SQL, understanding window functions is easy. Window functions, like aggregate functions, perform an aggregation on a defined set (a group) of rows, but rather than returning one value per group, window functions can return multiple values for each group. The group of rows to perform the aggregation on is the *window*. DB2 actually calls such functions *online analytic processing (OLAP) functions*, and Oracle calls them *analytic functions*, but the ISO SQL standard calls them window functions, so that's the term used in this book.

A Simple Example

Let's say that you want to count the total number of employees across all departments. The traditional method for doing that is to issue a COUNT(*) query against the entire EMP table:

```
select count(*) as cnt
  from emp

  CNT
-----
   14
```

This is easy enough, but often you will find yourself wanting to access such aggregate data from rows that do not represent an aggregation, or that represent a different aggregation. Window functions make light work of such problems. For example, the following query shows how you can use a window function to access aggregate data (the total count of employees) from detail rows (one per employee):

```
select ename,
       deptno,
       count(*) over() as cnt
  from emp
 order by 2

ENAME        DEPTNO    CNT
----------   ------ ------
CLARK            10     14
KING             10     14
MILLER           10     14
SMITH            20     14
ADAMS            20     14
FORD             20     14
SCOTT            20     14
JONES            20     14
ALLEN            30     14
BLAKE            30     14
MARTIN           30     14
JAMES            30     14
TURNER           30     14
WARD             30     14
```

The window function invocation in this example is COUNT(*) OVER(). The presence of the OVER keyword indicates that the invocation of COUNT will be treated as a window function, not as an aggregate function. In general, the SQL standard allows for all aggregate functions to also be window functions, and the keyword OVER is how the language distinguishes between the two uses.

So, what did the window function COUNT(*) OVER () do exactly? For every row being returned in the query, it returned the count of *all the rows* in the table. As the empty parentheses suggest, the OVER keyword accepts additional clauses to affect the range of rows that a given window function considers. Absent any such clauses, the window function looks at all rows in the result set, which is why you see the value 14 repeated in each row of output.

Hopefully you are beginning to see the great utility of window functions, which is that they allow you to work with multiple levels of aggregation in one row. As you continue through this appendix, you'll begin to see even more just how incredibly useful that ability can be.

Order of Evaluation

Before digging deeper into the OVER clause, it is important to note that window functions are performed as the last step in SQL processing prior to the ORDER BY clause. As an example of how window functions are processed last, let's take the query from the preceding section and use a WHERE clause to filter out employees from DEPTNO 20 and 30:

```
select ename,
       deptno,
       count(*) over() as cnt
  from emp
 where deptno = 10
 order by 2

ENAME      DEPTNO     CNT
---------- ------  ------
CLARK          10       3
KING           10       3
MILLER         10       3
```

The value for CNT for each row is no longer 14, it is now 3. In this example, it is the WHERE clause that restricts the result set to three rows; hence, the window function will count only three rows (there are only three rows available to the window function by the time processing reaches the SELECT portion of the query). From this example you can see that window functions perform their computations after clauses such as WHERE and GROUP BY are evaluated.

Partitions

Use the PARTITION BY clause to define a *partition* or group of rows to perform an aggregation over. As we've seen already, if you use empty parentheses, then the entire result set is the partition that a window function aggregation will be computed over. You can think of the PARTITION BY clause as a "moving GROUP BY" because unlike a traditional GROUP BY, a group created by PARTITION BY is not distinct in a result set. You can use PARTITION BY to compute an aggregation over a defined group of rows (resetting when a new group is encountered), and rather than having one group represent all instances of that value in the table, each value (each member in each group) is returned. Consider the following query:

```
select ename,
       deptno,
       count(*) over(partition by deptno) as cnt
  from emp
 order by 2


ENAME       DEPTNO      CNT
----------      ------   ------
CLARK           10        3
KING            10        3
MILLER          10        3
SMITH           20        5
ADAMS           20        5
FORD            20        5
SCOTT           20        5
JONES           20        5
```

```
ALLEN           30        6
BLAKE           30        6
MARTIN          30        6
JAMES           30        6
TURNER          30        6
WARD            30        6
```

This query still returns 14 rows, but now the COUNT is performed for each department as a result of the PARTITION BY DEPTNO clause. Each employee in the same department (in the same partition) will have the same value for CNT, because the aggregation will not reset (recompute) until a new department is encountered. Also note that you are returning information about each group, along with the members of each group. You can think of the preceding query as a more efficient version of the following:

```
select e.ename,
       e.deptno,
       (select count(*) from emp d
         where e.deptno=d.deptno) as cnt
  from emp e
 order by 2


ENAME        DEPTNO        CNT
---------- ------ ------
CLARK          10      3
KING           10      3
MILLER         10      3
SMITH          20      5
ADAMS          20      5
FORD           20      5
SCOTT          20      5
JONES          20      5
ALLEN          30      6
BLAKE          30      6
MARTIN         30      6
JAMES          30      6
TURNER         30      6
WARD           30      6
```

Additionally, what's nice about the PARTITION BY clause is that it performs its computations independently of other window functions, partitioning by different columns in the same SELECT statement. Consider the following query, which returns each employee, their department, the number of employees in their respective department, their job, and the number of employees with the same job:

```
select ename,
       deptno,
       count(*) over(partition by deptno) as dept_cnt,
       job,
       count(*) over(partition by job) as job_cnt
```

```
  from emp
 order by 2

ENAME          DEPTNO   DEPT_CNT JOB       JOB_CNT
----------   ------   -------- --------- -------
MILLER           10          3 CLERK           4
CLARK            10          3 MANAGER         3
KING             10          3 PRESIDENT       1
SCOTT            20          5 ANALYST         2
FORD             20          5 ANALYST         2
SMITH            20          5 CLERK           4
JONES            20          5 MANAGER         3
ADAMS            20          5 CLERK           4
JAMES            30          6 CLERK           4
MARTIN           30          6 SALESMAN        4
TURNER           30          6 SALESMAN        4
WARD             30          6 SALESMAN        4
ALLEN            30          6 SALESMAN        4
BLAKE            30          6 MANAGER         3
```

In this result set, you can see that employees in the same department have the same value for DEPT_CNT, and that employees who have the same job position have the same value for JOB_CNT.

By now it should be clear that the PARTITION BY clause works like a GROUP BY clause, but it does so without being affected by the other items in the SELECT clause and without requiring you to write a GROUP BY clause.

Effect of NULLs

Like the GROUP BY clause, the PARTITION BY clause lumps all the NULLs into one group or partition. Thus, the effect from NULLs when using PARTITION BY is similar to that from using GROUP BY. The following query uses a window function to count the number of employees with each distinct commission (returning –1 in place of NULL for readability):

```
select coalesce(comm,-1) as comm,
       count(*)over(partition by comm) as cnt
  from emp

  COMM        CNT
------ ----------
     0          1
   300          1
   500          1
  1400          1
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
```

```
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
```

Because COUNT(*) is used, the function counts rows. You can see that there are 10 employees having NULL commissions. Use COMM instead of *, however, and you get quite different results:

```
select coalesce(comm,-1) as comm,
       count(comm)over(partition by comm) as cnt
  from emp

COMM        CNT
---- ----------
   0          1
 300          1
 500          1
1400          1
  -1          0
  -1          0
  -1          0
  -1          0
  -1          0
  -1          0
  -1          0
  -1          0
  -1          0
  -1          0
```

This query uses COUNT(COMM), which means that only the non-NULL values in the COMM column are counted. There is one employee with a commission of 0, one employee with a commission of 300, and so forth. But notice the counts for those with NULL commissions! Those counts are 0. Why? Because aggregate functions ignore NULL values, or more accurately, aggregate functions count only non-NULL values.

When using COUNT, consider whether you want to include NULLs. Use COUNT(column) to avoid counting NULLs. Use COUNT(*) if you do want to include NULLs (since you are no longer counting actual column values, you are counting rows).

When Order Matters

Sometimes the order in which rows are treated by a window function is material to the results that you want to obtain from a query. For this reason, window function syntax includes an ORDER BY subclause that you can place within an OVER clause. Use the ORDER BY clause to specify how the rows are ordered with a partition

(remember, "partition" in the absence of a PARTITION BY clause means the entire result set).

Some window functions *require* you to impose order on the partitions of rows being affected. Thus, for some window functions, an ORDER BY clause is mandatory. At the time of this writing, SQL Server does not allow ORDER BY in the OVER clause when used with aggregate window functions. SQL Server does permit ORDER BY in the OVER clause when used with window ranking functions.

When you use an ORDER BY clause in the OVER clause of a window function, you are specifying two things:

- How the rows in the partition are ordered
- What rows are included in the computation

Consider the following query, which sums and computes a running total of salaries for employees in DEPTNO 10:

```
select deptno,
       ename,
       hiredate,
       sal,
       sum(sal)over(partition by deptno) as total1,
       sum(sal)over() as total2,
       sum(sal)over(order by hiredate) as running_total
  from emp
 where deptno=10

DEPTNO  ENAME   HIREDATE      SAL TOTAL1 TOTAL2 RUNNING_TOTAL
------  ------  ----------- ----- ------ ------ -------------
    10  CLARK   09-JUN-1981  2450   8750   8750          2450
    10  KING    17-NOV-1981  5000   8750   8750          7450
    10  MILLER  23-JAN-1982  1300   8750   8750          8750
```

Just to keep you on your toes, I've included a sum with empty parentheses. Notice how TOTAL1 and TOTAL2 have the same values. Why? Once again, the order in which window functions are evaluated answers the question. The WHERE clause filters the result set such that only salaries from DEPTNO 10 are considered for summation. In this case, there is only one partition—the entire result set, which consists of only salaries from DEPTNO 10. Thus TOTAL1, and TOTAL2 are the same.

Looking at the values returned by column SAL, you can easily see where the values for RUNNING_TOTAL come from. You can eyeball the values and add them

yourself to compute the running total. But more importantly, why did including an ORDER BY in the OVER clause create a running total in the first place? The reason is, when you use ORDER BY in the OVER clause, you are specifying a default "moving" or "sliding" window within the partition even though you don't see it. The ORDER BY HIREDATE clause terminates summation at the HIREDATE in the current row.

The following query is the same as the previous one, but uses the RANGE BETWEEN clause (which you'll learn more about later) to explicitly specify the default behavior that results from ORDER BY HIREDATE:

```
select deptno,
       ename,
       hiredate,
       sal,
       sum(sal)over(partition by deptno) as total1,
       sum(sal)over() as total2,
       sum(sal)over(order by hiredate
                    range between unbounded preceding
                      and current row) as running_total
  from emp
 where deptno=10

DEPTNO ENAME  HIREDATE      SAL TOTAL1 TOTAL2 RUNNING_TOTAL
------ ------ ----------- ----- ------ ------ -------------
    10 CLARK  09-JUN-1981  2450   8750   8750          2450
    10 KING   17-NOV-1981  5000   8750   8750          7450
    10 MILLER 23-JAN-1982  1300   8750   8750          8750
```

The RANGE BETWEEN clause that you see in this query is termed the *framing clause* by ANSI, and we'll use that term here. Now, it should be easy to see why specifying an ORDER BY in the OVER clause created a running total; we've (by default) told the query to sum all rows starting from the current row and include all prior rows ("prior" as defined in the ORDER BY, in this case ordering the rows by HIREDATE).

The Framing Clause

Let's apply the framing clause from the preceding query to the result set, starting with the first employee hired, who is named CLARK:

1. Starting with CLARK's salary, 2450, and including all employees hired before CLARK, compute a sum. Since CLARK was the first employee hired in DEPTNO 10, the sum is simply CLARK's salary, 2450, which is the first value returned by RUNNING_TOTAL.
2. Let's move to the next employee based on HIREDATE, named KING, and apply the framing clause once again. Compute a sum on SAL starting with the current row, 5000 (KING's salary), and include all prior rows (all employees hired before

KING). CLARK is the only one hired before KING, so the sum is 5000 + 2450, which is 7450, the second value returned by RUNNING_TOTAL.

3. Moving on to MILLER, the last employee in the partition based on HIREDATE, let's one more time apply the framing clause. Compute a sum on SAL starting with the current row, 1300 (MILLER's salary), and include all prior rows (all employees hired before MILLER). CLARK and KING were both hired before MILLER, and thus their salaries are included in MILLER's RUNNING_TOTAL: 2450 + 5000 + 1300 is 8750, which is the value for RUNNING_TOTAL for MILLER.

As you can see, it is really the framing clause that produces the running total. The ORDER BY defines the order of evaluation and happens to also imply a default framing.

In general, the framing clause allows you to define different "subwindows" of data to include in your computations. There are many ways to specify such subwindows. Consider the following query:

```
select deptno,
       ename,
       sal,
       sum(sal)over(order by hiredate
                    range between unbounded preceding
                      and current row) as run_total1,
       sum(sal)over(order by hiredate
                     rows between 1 preceding
                     and current row) as run_total2,
       sum(sal)over(order by hiredate
                    range between current row
                     and unbounded following) as run_total3,
       sum(sal)over(order by hiredate
                     rows between current row
                     and 1 following) as run_total4
  from emp
 where deptno=10

DEPTNO ENAME    SAL RUN_TOTAL1 RUN_TOTAL2 RUN_TOTAL3 RUN_TOTAL4
------ ------ ----- ---------- ---------- ---------- ----------
    10 CLARK   2450       2450       2450       8750       7450
    10 KING    5000       7450       7450       6300       6300
    10 MILLER  1300       8750       6300       1300       1300
```

Don't be intimidated here; this query is not as bad as it looks. You've already seen RUN_TOTAL1 and the effects of the framing clause UNBOUNDED PRECEDING AND CURRENT ROW. Here's a quick description of what's happening in the other examples:

RUN_TOTAL2

Rather than the keyword RANGE, this framing clause specifies ROWS, which means the *frame*, or window, is going to be constructed by counting some number of rows. The 1 PRECEDING means that the frame will begin with the row immediately preceding the current row. The range continues through the CURRENT ROW. So what you get in RUN_TOTAL2 is the sum of the current employee's salary and that of the preceding employee, based on HIREDATE.

```
[[sqlckbk-APP-A-NOTE-11]]
```

It so happens that RUN_TOTAL1 and RUN_TOTAL2 are the same for both CLARK and KING. Why? Think about which values are being summed for each of those employees, for each of the two window functions. Think carefully, and you'll get the answer.

RUN_TOTAL3

The window function for RUN_TOTAL3 works just the opposite of that for RUN_TOTAL1; rather than starting with the current row and including all prior rows in the summation, summation begins with the current row and includes all subsequent rows in the summation.

RUN_TOTAL4

This is the inverse of RUN_TOTAL2; rather than starting from the current row and including one prior row in the summation, start with the current row and include one subsequent row in the summation.

If you can understand what's been explained thus far, you will have no problem with any of the recipes in this book. If you're not catching on, though, try practicing with your own examples and your own data. It's usually easier to learn by coding new features rather than just reading about them.

A Framing Finale

As a final example of the effect of the framing clause on query output, consider the following query:

```
select ename,
       sal,
       min(sal)over(order by sal) min1,
       max(sal)over(order by sal) max1,
       min(sal)over(order by sal
                    range between unbounded preceding
                      and unbounded following) min2,
       max(sal)over(order by sal
```

```
                    range between unbounded preceding
                      and unbounded following) max2,
       min(sal)over(order by sal
                    range between current row
                      and current row) min3,
       max(sal)over(order by sal
                    range between current row
                      and current row) max3,
       max(sal)over(order by sal
                     rows between 3 preceding
                      and 3 following) max4
  from emp

ENAME      SAL     MIN1    MAX1    MIN2    MAX2    MIN3    MAX3    MAX4
------  -----   ------  ------  ------  ------  ------  ------  ------
SMITH      800      800     800     800    5000     800     800    1250
JAMES      950      800     950     800    5000     950     950    1250
ADAMS     1100      800    1100     800    5000    1100    1100    1300
WARD      1250      800    1250     800    5000    1250    1250    1500
MARTIN    1250      800    1250     800    5000    1250    1250    1600
MILLER    1300      800    1300     800    5000    1300    1300    2450
TURNER    1500      800    1500     800    5000    1500    1500    2850
ALLEN     1600      800    1600     800    5000    1600    1600    2975
CLARK     2450      800    2450     800    5000    2450    2450    3000
BLAKE     2850      800    2850     800    5000    2850    2850    3000
JONES     2975      800    2975     800    5000    2975    2975    5000
SCOTT     3000      800    3000     800    5000    3000    3000    5000
FORD      3000      800    3000     800    5000    3000    3000    5000
KING      5000      800    5000     800    5000    5000    5000    5000
```

OK, let's break this query down:

MIN1

The window function generating this column does not specify a framing clause, so the default framing clause of UNBOUNDED PRECEDING AND CURRENT ROW kicks in. Why is MIN1 800 for all rows? It's because the lowest salary comes first (ORDER BY SAL), and it remains the lowest, or minimum, salary forever after.

MAX1

The values for MAX1 are much different from those for MIN1. Why? The answer (again) is the default framing clause UNBOUNDED PRECEDING AND CURRENT ROW. In conjunction with ORDER BY SAL, this framing clause ensures that the maximum salary will also correspond to that of the current row.

Consider the first row, for SMITH. When evaluating SMITH's salary and all prior salaries, MAX1 for SMITH is SMITH's salary, because there are no prior salaries. Moving on to the next row, JAMES, when comparing JAMES's salary to all prior salaries, in this case comparing to the salary of SMITH, JAMES's salary is the

higher of the two, and thus it is the maximum. If you apply this logic to all rows, you will see that the value of MAX1 for each row is the current employee's salary.

MIN2 and MAX2
: The framing clause given for these is UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING, which is the same as specifying empty parentheses. Thus, all rows in the result set are considered when computing MIN and MAX. As you might expect, the MIN and MAX values for the entire result set are constant, and thus the value of these columns is constant as well.

MIN3 and MAX3
: The framing clause for these is CURRENT ROW AND CURRENT ROW, which simply means use only the current employee's salary when looking for the MIN and MAX salary. Thus, both MIN3 and MAX3 are the same as SAL for each row. That was easy, wasn't it?

MAX4
: The framing clause defined for MAX4 is 3 PRECEDING AND 3 FOLLOWING, which means, for every row, consider the three rows prior and the three rows after the current row, as well as the current row itself. This particular invocation of MAX(SAL) will return from those rows the highest salary value.

  If you look at the value of MAX4 for employee MARTIN, you can see how the framing clause is applied. MARTIN's salary is 1250, and the three employee salaries prior to MARTIN's are WARD's (1250), ADAMS's (1100) and JAMES's (950). The three employee salaries after MARTIN's are MILLER's (1300), TURNER's (1500), and ALLEN's (1600). Out of all those salaries, including MARTIN's, the highest is ALLEN's, and thus the value of MAX4 for MARTIN is 1600.

Readability + Performance = Power

As you can see, window functions are extremely powerful as they allow you to write queries that contain both detailed and aggregate information. Using window functions allows you to write smaller, more efficient queries as compared to using multiple self-join and/or scalar subqueries. Consider the following query, which easily answers all of the following questions: "What is the number of employees in each department? How many different types of employees are in each department (e.g., how many clerks are in department 10)? How many total employees are in table EMP?"

```
select deptno,
       job,
       count(*) over (partition by deptno) as emp_cnt,
       count(job) over (partition by deptno,job) as job_cnt,
       count(*) over () as total
  from emp
```

```
DEPTNO JOB          EMP_CNT    JOB_CNT      TOTAL
------ --------- ---------- ---------- ----------
    10 CLERK              3          1         14
    10 MANAGER            3          1         14
    10 PRESIDENT          3          1         14
    20 ANALYST            5          2         14
    20 ANALYST            5          2         14
    20 CLERK              5          2         14
    20 CLERK              5          2         14
    20 MANAGER            5          1         14
    30 CLERK              6          1         14
    30 MANAGER            6          1         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
```

Returning the same result set without using window functions would require a bit more work:

```
select a.deptno, a.job,
       (select count(*) from emp b
         where b.deptno = a.deptno) as emp_cnt,
       (select count(*) from emp b
         where b.deptno = a.deptno and b.job = a.job) as job_cnt,
       (select count(*) from emp) as total
  from emp a
 order by 1,2

DEPTNO JOB          EMP_CNT     JOB_CNT      TOTAL
------ --------- ----------  ---------- ----------
    10 CLERK              3           1         14
    10 MANAGER            3           1         14
    10 PRESIDENT          3           1         14
    20 ANALYST            5           2         14
    20 ANALYST            5           2         14
    20 CLERK              5           2         14
    20 CLERK              5           2         14
    20 MANAGER            5           1         14
    30 CLERK              6           1         14
    30 MANAGER            6           1         14
    30 SALESMAN           6           4         14
    30 SALESMAN           6           4         14
    30 SALESMAN           6           4         14
    30 SALESMAN           6           4         14
```

The nonwindow solution is obviously not difficult to write, yet it certainly is not as clean or efficient (you won't see performance differences with a 14-row table, but try these queries with, say, a 1,000- or 10,000-row table, and then you'll see the benefit of using window functions over multiple self-joins and scalar subqueries).

Providing a Base

Besides readability and performance, window functions are useful for providing a "base" for more complex "report-style" queries. For example, consider the following "report-style" query that uses window functions in an inline view and then aggregates the results in an outer query. Using window functions allows you to return detailed as well as aggregate data, which is useful for reports. The following query uses window functions to find counts using different partitions. Because the aggregation is applied to multiple rows, the inline view returns all rows from EMP, which the outer CASE expressions can use to transpose and create a formatted report:

```
select deptno,
       emp_cnt as dept_total,
       total,
       max(case when job = 'CLERK'
                then job_cnt else 0 end) as clerks,
       max(case when job = 'MANAGER'
                then job_cnt else 0 end) as mgrs,
       max(case when job = 'PRESIDENT'
                then job_cnt else 0 end) as prez,
       max(case when job = 'ANALYST'
                then job_cnt else 0 end) as anals,
       max(case when job = 'SALESMAN'
                then job_cnt else 0 end) as smen
  from (
select deptno,
       job,
       count(*) over (partition by deptno) as emp_cnt,
       count(job) over (partition by deptno,job) as job_cnt,
       count(*) over () as total
  from emp
       ) x
 group by deptno, emp_cnt, total

DEPTNO DEPT_TOTAL TOTAL CLERKS MGRS PREZ ANALS SMEN
------ ---------- ----- ------ ---- ---- ----- ----
    10          3    14      1    1    1     0    0
    20          5    14      2    1    0     2    0
    30          6    14      1    1    0     0    4
```

The previous query returns each department, the total number of employees in each department, the total number of employees in table EMP, and a breakdown of the number of different job types in each department. All this is done in one query, without additional joins or temp tables!

As a final example of how easily multiple questions can be answered using window functions, consider the following query:

```
select ename as name,
       sal,
       max(sal)over(partition by deptno) as hiDpt,
       min(sal)over(partition by deptno) as loDpt,
       max(sal)over(partition by job) as hiJob,
       min(sal)over(partition by job) as loJob,
       max(sal)over() as hi,
       min(sal)over() as lo,
       sum(sal)over(partition by deptno
                        order by sal,empno) as dptRT,
       sum(sal)over(partition by deptno) as dptSum,
       sum(sal)over() as ttl
  from emp
 order by deptno,dptRT
```

```
NAME     SAL HIDPT LODPT HIJOB LOJOB    HI   LO DPTRT DPTSUM     TTL
------ ----- ----- ----- ----- ----- ----- ---- ------ ------ ------
MILLER  1300  5000  1300  1300   800  5000  800   1300   8750  29025
CLARK   2450  5000  1300  2975  2450  5000  800   3750   8750  29025
KING    5000  5000  1300  5000  5000  5000  800   8750   8750  29025
SMITH    800  3000   800  1300   800  5000  800    800  10875  29025
ADAMS   1100  3000   800  1300   800  5000  800   1900  10875  29025
JONES   2975  3000   800  2975  2450  5000  800   4875  10875  29025
SCOTT   3000  3000   800  3000  3000  5000  800   7875  10875  29025
FORD    3000  3000   800  3000  3000  5000  800  10875  10875  29025
JAMES    950  2850   950  1300   800  5000  800    950   9400  29025
WARD    1250  2850   950  1600  1250  5000  800   2200   9400  29025
MARTIN  1250  2850   950  1600  1250  5000  800   3450   9400  29025
TURNER  1500  2850   950  1600  1250  5000  800   4950   9400  29025
ALLEN   1600  2850   950  1600  1250  5000  800   6550   9400  29025
BLAKE   2850  2850   950  2975  2450  5000  800   9400   9400  29025
```

This query answers the following questions easily, efficiently, and readably (and without additional joins to EMP!). Simply match the employee and their salary with the different rows in the result set to determine:

- Who makes the highest salary of all employees (HI)
- Who makes the lowest salary of all employees (LO)
- Who makes the highest salary in the department (HIDPT)
- Who makes the lowest salary in the department (LODPT)
- Who makes the highest salary in their job (HIJOB)
- Who makes the lowest salary in their job (LOJOB)
- What is the sum of all salaries (TTL)
- What is the sum of salaries per department (DPTSUM)
- What is the running total of all salaries per department (DPTRT)

APPENDIX B
Common Table Expressions

Many of the queries presented in this cookbook go beyond what is possible using tables as they are typically available in a database, especially in relation to aggregate functions and window functions. Therefore, for some queries, you need to make a derived table—either a subquery or a common table expression (CTE).

Subqueries

Arguably the simplest way to create a virtual table that allows you to run queries on window functions or aggregate functions is a subquery. All that's required here is to write the query that you need within parentheses and then to write a second query that uses it. The following table illustrates the use of subqueries with a simple *double aggregate*—you want to find not just the counts of employees in each job, but then identify the highest number, but you can't nest aggregate functions directly in a standard query.

One pitfall is that some vendors require you to give the subquery table and alias, but others do not. The following example was written in MySQL, which does require an alias. The alias here is HEAD_COUNT_TAB after the closing parenthesis.

Others that require an alias are PostgreSQL and SQL Server, while Oracle does not:

```
select max(HeadCount) as HighestJobHeadCount from
(select job,count(empno) as HeadCount
from emp
group by job) head_count_tab
```

Common Table Expressions

CTEs were intended to overcome some of the limits of subqueries, and may be most well known for allowing recursive queries to be used within SQL. In fact, enablng recursion within SQL was the main inspiration for CTEs.

This example achieves the same result as the subquery we saw earlier—it finds a *double aggregate*:

```
with head_count_tab (job,HeadCount) as

(select job,count(empno)
from emp
group by job)

select max(HeadCount) as HighestJobHeadCount
from head_count_tab
```

Although this query solves a simple problem, it illustrates the essential features of a CTE. We introduce the derived table using the WITH clause, specifying the column headings in the parentheses, and use parentheses around the derived table's query itself. If we want to add more derived tables, we can add more as long as we separate each one with a comma and provide its name before its query (the reverse of how aliasing usually works in SQL).

Because the inner queries are presented before the outer query, in many circumstances they may also be considered more readable—they make it easier to study each logical element of the query separately in order to understand the logical flow. Of course, as with all things in coding, this will vary according to circumstances, and sometimes the subquery will be more readable.

Considering that recursion is the key reason for CTEs to exist, the best way to demonstrate their capability is through a recursive query.

The query that follows calculates the first 20 numbers in the Fibonacci sequence using a recursive CTE. Note that in the first part of the anchor query, we can initialize the values in the first row of the virtual table:

```
with recursive workingTable ( fibNum, NextNumber, index1)
as
(select 0,1,1
union all
select fibNum+nextNumber,fibNUm,index1+1
from anchor
where index1<20)

select fibNum from workingTable as fib
```

The Fibonacci sequence finds the next number by adding the current and previous numbers; you could also use LAG to achieve this result. However, in this case we've made a pseudo-LAG by using two columns to account for the current number and the previous. Note the keyword RECURSIVE, which is mandatory in MySQL, Oracle, and PostgreSQL but not in SQL Server or DB2. In this query, the index1 column is largely redundant in the sense of not being used for the Fibonacci calculation. Instead, we have included it to make it simpler to set the number of rows returned via the WHERE clause. In a recursive CTE, the WHERE clause becomes crucial, as without it the query would not terminate (although in this specific case, if you try deleting it, you are likely to find that your database throws an overflow error when the numbers become too large for the data type).

At the simple end of the spectrum, there's not a lot of difference between a subquery and CTE in terms of usability. Both allow for nesting or writing more complicated queries that refer to other derived tables. However, once you start nesting many subqueries, readability is lessened because the meaning of different variables is hidden in successive query layers. In contrast, because a CTE arranges each element vertically, it is easier to understand the meaning of each element.

Summing Up

The use of derived tables dramatically extends the range of SQL. Both subqueries and CTES are used many times throughout the book, so it is important to understand how they work, especially as they each have a particular syntax that you need to master to ensure success. The recursive CTE, which is now available in the vendor offerings in this book, is one of the biggest extensions to have occurred within SQL, allowing for many extra possibilities.

Index

Symbols

% (modulus) function (SQL Server), 288
% (wildcard) operator, 13
* character in SELECT statements, 1
+ (concatenation) operator (SQL Server), 7, 307
_ (underscore) operator, 13
|| (concatenation) function (DB2/Oracle/PostgreSQL), 7, 306

A

abstraction, axiom of, 511
ADDDATE function (MySQL), 247, 262, 292, 299
ADD_MONTHS function (Oracle), 282, 285, 298
aggregate functions
  multiple tables and, 52-59
  NULL values and, 190, 512
  WHERE clause, referencing in, 5
aliases
  for CASE expression, 8
  inline views, 495
  referencing aliased columns, 5
alphabetizing strings, 141-146
alphanumeric strings
  converting to numbers, 193-195
  determining whether a string is alphanumeric, 116-120
  mixed, 472-473
  sorting mixed, 18-21
anti-joins, 31
AS keyword, 4
asterisk (*) character in SELECT statements, 1
asterisk (*) character with COUNT function, 175, 177
averages, computing, 169-171
AVG function, 169-171
axiom of abstraction, 511
axiom of specification, 512
axiom schema of separation, 512
axiom schema of subsets, 512

B

Barber Puzzle, 511
Benford's law, 201-203
binary, converting whole numbers to, 474-477

C

calendars, creating, 268-280
Cartesian products, 51, 106
CASE expression, 8, 12, 23, 28, 188, 196
CAST function (SQL Server), 293
CEIL function (DB2/MySQL/Oracle/PostgreSQL), 388
CEILING function (SQL Server), 388, 417
COALESCE function, 12, 64, 170, 190, 246, 319
columns
  adding headers to double pivoted result sets, 481-491
  concatenating, 6
  naming, 4
  retrieving a subset of columns from a table, 3
  retrieving all rows and columns from a table, 1
common table expressions (CTEs), xi, 536

composite subqueries, converting scalar subqueries to (Oracle), 493-495
CONCAT function (MySQL), 7, 121, 307
concatenation
column values, 6
operator (+) (SQL Server), 7, 307
operator (||) (DB2/Oracle/PostgreSQL), 7, 306
CONCAT_WS function (MySQL), 121, 124
conditional logic in SELECT statements, 7
CONNECT BY clause (Oracle), 269, 295
in hierarchical structures, 446, 450
WITH clause and, 360
CONNECT_BY_ISLEAF function (Oracle), 451, 455
CONNECT_BY_ROOT function (Oracle), 451, 455
constraints, listing, 95
correlated subquery, 39
COUNT function, 88, 175-178, 512
COUNT OVER window function, 420
count star, 177
create table as select (CTAS), 72
CREATE TABLE command, 71
CREATE TABLE … LIKE command (DB2), 72
cross-tab reports
creating (SQL Server), 459-461
unpivoting (SQL Server), 461-463
CTAS (create table as select), 72
CTEs (common table expressions), xi, 536
CUBE extension, 401, 406
CUME_DIST function, 186
CURRENT_DATE function (DB2/MySQL/PostgreSQL), 267, 362

D

data dependent keys, sorting on, 27
data dictionary views (Oracle), 102
DATE function (DB2), 289
date manipulation, 239-311
comparing records using specific parts of a date, 302-305
creating a calendar, 268-280
determining all dates for a particular weekday throughout a year, 255-260
determining quarter start/end dates for a, 286-293
determining the date of first/last occurrences of specific weekday in month, 261-268
determining the first/last days of a month, 252-254
determining the number of days in a year, 246-249, 470
determining whether a year is a leap year, 240-246
extracting units of time from date, 249-252
filling in missing dates, 293-301
identifying overlapping date ranges, 305-310
listing quarter/end dates for the year, 281-286
searching on specific units of time, 301-302
DATEADD function (SQL Server), 247, 253
DATEDIFF function (MySQL/SQL Server), 247
DATENAME function (SQL Server), 302, 304
DATEPART function (SQL Server), 249, 251, 260, 283, 286
dates, ORDER BY clause and (DB2), 422
DATE_FORMAT function (MySQL), 251, 303
DATE_TRUNC function (PostgreSQL), 243, 247, 253
DAY function (DB2), 246, 250, 252
DAY function (MySQL), 246, 253
DAY function (SQL Server), 246, 253
DAYNAME function (DB2/MySQL/SQL Server), 301
DAYOFWEEK function (DB2/MYSQL), 267, 303
DAYOFYEAR function (DB2/MySQL/SQL Server), 246-248, 297, 299-301
DAYS function (DB2), 246
DECODE function (Oracle), 504
DEFAULT keyword, 69
DEFAULT VALUES clause (PostgreSQL/SQL Server), 69
DELETE command, 81, 83
deleting records
all, 83
duplicate, 85-87
with NULLs (PostgreSQL/MySQL), 374
with NULLs (DB2/Oracle/SQL Server), 373
referenced from another table, 87-88
referencing nonexistent records from another table, 85
referential integrity violations, 85

single, 84
specific, 83
delimited data, converting to IN-list, 136-141
delimited lists, creating, 132-135
DENSE_RANK function (DB2/Oracle/SQL Server), 343, 355, 357
DENSE_RANK OVER window function (DB2/Oracle/SQL Server), 343, 350, 478
DENSE_RANK window function, 183
DEPT table structure, xiii
DICTIONARY view, 102
DISTINCT keyword
alternatives to, 31, 351
SELECT list and, 17, 352, 510
uses for, 36, 54, 343
double aggregate, 535
duplicates
deleting, 85-87
suppressing, 351-353
dynamic SQL, creating, 100-102

E

EMP table structure, xiii
equi-join operations, 31, 40
EXCEPT function, 35, 45, 48
EXTRACT function (PostgreSQL/MySQL), 327
extreme values, finding, 344

F

FETCH FIRST clause (DB2), 8
Fibonacci sequence, 536
forecasts, generating simple, 359-367
foreign keys, listing, 97-100
framing clause, 483
Frege's axiom, 511
Frege, Gottlob, 511
FULL OUTER JOIN command, 62

G

GENERATE_SERIES function (PostgreSQL)
parameters, 332
uses, 241, 244, 270, 331
GETDATE function (SQL Server), 362
GROUP BY clause, 353, 507
GROUP BY queries, returning other columns in, 394-396
grouping, 507-519
COUNT function and, 175
defined, 508-512
SELECT clause and, 173, 516-519
SUM function and, 173
testing for existence of a value within a group, 502-504
by time units, 416-419
GROUPING function (DB2/Oracle/SQL Server), 398, 406, 432
GROUPING SETS extension (DB2/Oracle), 406-408
GROUP_CONCAT function, 133, 143, 145

H

hierarchical queries, 435-458
creating hierarchical view of a table, 444-448
determining which rows are leaf/branch/root nodes, 450-458
expressing a child-parent-grandparent relationship, 440-444
expressing a parent-child relationship, 436-439
finding all child rows for given parent row, 449
histograms
horizontal, 390-391
vertical, 392-393
HOUR function (DB2), 250

I

IF-ELSE operations, 7
implicit type conversion, 20
IN-lists, converting delimited data into, 136-141
indexes, listing, 94
information schema (MySQL/PostgreSQL/SQL Server), 91
inline views
naming, 495
referencing aliased columns with, 5
inner joins, 31, 340
INSERT ALL statement (Oracle), 73
INSERT FIRST statement (Oracle), 73
INSERT statement, 68, 70
inserting into a column, 70
inserting records
blocking, 74
copying rows from one table to another, 70
with default values, 68-70

into multiple tables, 72-74
new records, 68
with NULL values, 70
INSTR function, 140, 155, 159
INSTR function (Oracle), 468
integrity, deleting records violating, 85
INTERSECT operation, 33-34
ISNUMERIC function, 150
ITERATE command (Oracle), 331
ITERATION_NUMBER function (Oracle), 332

J

JOIN clause, 33
joins
about, 31
aggregates and, 52-57
anti-, 31
equi-, 31, 40
inner, 31, 340
scalar subqueries and, 43
selecting columns, 34
self-, 307, 342

K

KEEP extension (Oracle), 183, 184, 357
keys
data dependent, 27
foreign, 97-100
preserving, 80
knight values, 353-358
Kyte, Tom, 475

L

LAG OVER window function (Oracle), 316-325, 348-349, 383
LAG window function, 182
LAST function (Oracle), 355, 357
LAST_DAY function (MySQL/Oracle), 240, 241, 252, 253, 266
LEAD OVER window function (Oracle)
default behavior, 321
duplicates and, 319
options, 321, 346, 349
self-joins and, 308-310, 314, 317
uses, 315, 347, 348
leap years, 240-246
LEN function, 126
LENGTH function, 109, 126
LIKE operator, 13
LIMIT clause (MySQL/PostgreSQL), 8, 10
LIST_AGG function, 133
logarithms, 180
loop functionality limits, in SQL, 105
LPAD function (Oracle/PostgreSQL/MySQL), 390
LTRIM function (Oracle), 444

M

matrices, creating sparse, 414
MAX function, 171, 183, 324
MAX OVER window function, 344, 349
maximum values, finding, 171-173
median absolute deviation, finding outliers with, 197-201
MEDIAN function (Oracle), 186
medians, calculating, 185-187
MERGE statement, 67, 82
merging records, 81-83
metadata queries, 91-103
describing data dictionary views in an Oracle database, 102
listing a table's columns, 93
listing constraints on a table, 95
listing foreign keys without corresponding indexes, 97-100
listing indexed columns for a table, 94
listing tables in a schema, 91
using SQL to generate SQL, 100-102
MIN function, 171, 324
MIN OVER window function (DB2/Oracle/SQL Server), 328, 344, 349
minimum values, finding, 171-173
MINUS operation, 35, 36, 45, 48
MINUTE function (DB2), 250
MODEL clause (Oracle), 331, 463-467, 474-477
modes, calculating, 182-185
modifying records
changing row data, 75
copying rows from one table to another, 70
modifying values in a table, 75
using queries for new values, 80
with values from another table, 78-81
when corresponding rows exist, 77
modulus (%) function (SQL Server), 288
MONTH function (DB2/MySQL), 242, 250, 268

MONTHNAME function (DB2/MySQL), 301, 303
multiple tables, inserting data into, 72-74
multiple tables, retrieving data from, 29-65
adding joins to a query without interfering with other joins, 42-44
Cartesian products and, 51
combining related rows, 31-33
comparing, 44-51
finding rows in common between two tables, 33
joins when aggregates are used, 52-57
nonmatching rows, 40
NULLs in operations/comparisons, 64
outer joins when using aggregates, 57-59
retrieving rows from one table that do not correspond to rows in another, 40
retrieving values from one table that do not exist in another, 34-40
returning missing data from multiple tables, 60-63
stacking one rowset atop another, 29-31

N

n-1 rule, 52
names, extracting initials from, 120-124
new records, inserting, 68
NEWID function, 11
NEXT_DAY function (Oracle), 265-266
NOT EXISTS, 85
NOT IN operator, 36
NROWS function (DB2/SQL Server), 362
NTILE window function (Oracle/SQL Server), 389
NULL paradox, 512
NULLs
aggregate functions and, 190
AVG function and, 170
comparisons to, 439
COUNT function and, 177
finding null values, 11
inserting records with, 70
MIN/MAX functions and, 172
NOT IN operator and, 36
OR operations and, 36
overriding a default value with, 70
removing (DB2/Oracle/SQL Server), 373
removing (PostgreSQL/MySQL), 374
sorting and, 21-27
SUM function and, 174, 176
transforming into real values, 12
window functions and, 523
NULLS FIRST extension, 25
NULLS LAST extension, 25
numbers queries, 169-203
aggregating nullable columns, 190
averages, 169-171
averages without high/low values, 191-193
calculating a median, 185-187
calculating a mode, 182-185
changing values in a running total, 196-197
converting alphanumeric strings into numbers, 193-195
converting whole to binary (Oracle), 474-477
counting rows in a table, 175-177
counting values in a column, 177
determining the percentage of a total, 187-190
finding anomalies using Benford's law, 201-203
finding outliers using the median absolute deviation, 197-201
finding the min/max value in a column, 171-173
generating a running product, 179
generating a running total, 178
percentage relative to total, 500-501
smoothing a series of values, 181
subtotals for all combinations, 400-410
subtotals, simple, 397-400
summing values in a column, 173-175

O

ORDER BY clause, 10, 15, 17, 178, 422
(see also sorting query results)
outer joins
OR logic in, 341
Oracle syntax, 43, 295
when using aggregates, 57-59
outliers, median absolute deviation for finding, 197-201
OVER keyword, 26

P

PARTITION BY clause, 521-523
patterns, searches for matching, 13
percent (%) operator, 13

percentage calculations, 187-190, 500-501
PERCENTILE_CONT function, 185-187, 198
PIVOT operator (SQL Server), 459-461
pivot tables, xiv
pivoting
about, 370
inter-row calculations, 384-386
MODEL clause (Oracle), 463-467
multiple rows, results into, 372-377
one row, results into, 369-371
ranked result sets, 477-481
reverse, 377-378
subtotals, result sets with, 429-434
PRIOR keyword (Oracle), 443

Q

QUARTER function (DB2/MySQL), 285
quotes, embedding within string literals, 108

R

RAND function, 10
RANDOM function, 11
random records, retrieving, 10
ranges, 313-333
filling in missing values, 326-329
finding differences between rows in same group/partition, 317-323
generating consecutive numeric values, 330-333
locating range of consecutive values, 313-316
locating the beginning/end of a range of consecutive values, 323-326
RANK OVER window function, 350
RATIO_TO_REPORT function (Oracle), 501
reciprocal rows, searching for, 341-343
records
merging, 81-83
sorting (see sorting query results)
RECURSIVE keyword, 537
referential integrity, deleting records violating, 85
REGEXP_REPLACE function, 166
REPEAT function, 121
REPEAT function (DB2), 390
REPLACE function, 20, 105
(see also strings)
REPLICATE function (SQL Server), 390
reports, queries for creating, 369-434
calculating simple subtotals, 397-400
calculating subtotals for all possible expression combinations, 400-410
creating a predefined number of buckets, 388
creating a sparse matrix, 414
creating buckets of data, of a fixed size, 386-388
creating horizontal histograms, 390-391
creating vertical histograms, 392-393
grouping rows by units of time, 416-419
identifying rows that are not subtotals, 410-412
performing aggregations over a moving range of values, 422-429
performing aggregations over different groups/partitions simultaneously, 420-421
pivoting a result set into multiple rows, 372-377
pivoting a result set into one row, 369-371
pivoting a result set to facilitate inter-row calculations, 384-386
pivoting a result set with subtotals, 429-434
returning non-GROUP BY columns, 394-396
reverse pivoting a result set, 377-378
reverse pivoting a result set into one column, 379-381
suppressing repeating values from a result set, 382-384
using case expressions to flag rows, 412-414
result set, transposing (Oracle), 463-467
retrieving records, 1-14
concatenating column values, 6
finding null values, 11
finding rows that satisfy multiple conditions, 2
limiting the number of rows returned, 8-10
providing meaningful names for columns, 4
referencing an aliased column in the WHERE clause, 5
for reports (see reports, queries for creating)
retrieving a subset of columns from a table, 3
retrieving a subset of rows from a table, 2
retrieving all rows and columns from a table, 1
returning n random records from a table, 10

searching for patterns, 13
transforming nulls into real values, 12
using conditional logic in a SELECT statement, 7
reverse pivoting result sets, 377-378
robust statistics, 193
ROLLUP extension of GROUP BY (DB2/Oracle), 397, 410, 430
row generation, dynamic, 330
ROWNUM function (Oracle), 9, 285, 337
rows
copying from one table to another, 70
finding rows that satisfy multiple conditions, 2
limiting the number of rows returned, 8-10
parsing serialized data into, 495-499
retrieving a subset of rows from a table, 2
retrieving all rows and columns from a table, 1
ROW_NUMBER function (Oracle), 135
ROW_NUMBER function (SQL Server), 144
ROW_NUMBER OVER window function (DB2/Oracle/SQL Server)
ORDER BY clause and, 352
uniqueness of result, 336
uses, 284, 336-339, 351, 373
RPAD function, 122, 480
RTRIM function (Oracle/PostgreSQL), 301
RULES subclause (Oracle), 465
running products, 179
running totals, 178, 196-197
Russell's Paradox, 511
Russell, Bertrand, 511

S

scalar subqueries
converting to composite (Oracle), 493-495
joins and, 43
referencing in WHERE clause, 10
scripts, generating, 100-102
searching, 335-367
determining which rows are reciprocals, 341-343
finding knight values, 353-358
finding records with highest/lowest values, 344
generating simple forecasts, 359-367
incorporating OR logic when using outer joins, 339-341
investigating future rows, 345-347
paginating through a result set, 335-337
patterns, 13
ranking results, 350
selecting top n records, 343
shifting row values, 347-349
skipping n rows from a table, 338
suppressing duplicates, 351-353
SECOND function (DB2), 250
SELECT function, 108
SELECT statements, 1
(see also retrieving records)
* character in, 1
conditional logic in, 7
DISTINCT keyword and, 17, 352, 510
GROUP BY and, 173, 516-519
partial, xiv
self-joins
alternatives to, 308, 314, 324
uses, 307, 342
separation, axiom schema of, 512
serialized data, parsing into rows, 495-499
SET differences, 45
SHOW INDEX command, 94, 99
SIGN function (MySQL/PostgreSQL), 266
sorting query results, 15-28
on data-dependent key, 27
mixed alphanumeric data, 18-21
by multiple fields, 16
NULLS and, 21-27
returning in a specified order, 15-16
by substrings, 17
SOUNDEX function, 163
specification, axiom of, 512
SPLIT_PART function, 137, 141, 155, 161
star (*) character in SELECT statements, 1
START WITH clause (Oracle), 447, 449
Stoll, Robert, 511
strings, 105-167
alphabetizing, 141-146
alphanumeric, sorting mixed, 18-21
comparing strings by sound, 162-164
converting alphanumeric strings to numbers, 193-195
converting delimited data into a multivalued IN-list, 136-141
counting the occurrences of a character in a string, 109

creating a delimited list from table rows, 132-135
determining whether alphanumeric, 116-120
embedding quotes within string literals, 108
extracting elements from unfixed locations, 467-470
extracting initials from a name, 120-124
extracting the nth delimited substring, 153-160
finding text not matching a pattern, 164-167
identifying strings that can be treated as numbers, 147-153
mixed alphanumeric, 472-473
ordering by a number in a string, 126-132
ordering by parts of a string, 125-126
parsing an IP address, 160-162
parsing into rows, 495-499
removing unwanted characters from, 110-112
separating numeric and character data, 112-116
traversing, 106-108
walking a string, 106-108
STRING_AGG function, 134, 143, 155
STRING_SPLIT function, 155, 158
STR_TO_DATE function (MySQL), 292
subqueries, 493-495, 535
subsets, axiom schema of, 512
SUBSTR function (DB2/MySQL/Oracle/PostgreSQL), 18, 124, 126, 140, 144, 146, 155, 159
SUBSTRING function (MySQL), 121
SUBSTRING function (SQL Server), 18, 126, 144, 288, 292
substrings
extracting the nth delimited substring, 153-160
sorting query results by, 17
SUBSTRING_INDEX function (MySQL), 121, 124, 139, 161
subtotals
calculating for all combinations, 400-410
calculating simple, 397-400
pivoting result set with, 429-434
SUM function, 174, 176
SUM OVER window function (DB2/Oracle/SQL Server), 55, 59, 178, 180, 188, 196
summing column values, 173-175
SYS_CONNECT_BY_PATH function (Oracle), 133, 135, 143, 145, 441, 444, 457

T

tables, creating with same columns as existing table, 71
time, grouping rows by, 416-419
TOP keyword (SQL Server), 9
TO_BASE function (Oracle), 475
TO_CHAR function (Oracle/PostgreSQL), 244, 245, 250, 255, 259
TO_NUMBER function (Oracle/PostgreSQL), 250
TRANSLATE function, 20, 105
(see also strings)
transposing result sets (Oracle), 463-467
trimmed mean, 191, 193
TRUNC function (Oracle), 248, 252, 285
TRUNCATE command, 83

U

underscore (_) operator, 13
UNION ALL operation, 29-31, 46, 48-51, 73, 516
UNION operation, 30, 48
UNPIVOT operator (SQL Server), 461-463
UPDATE statement, 76-83

V

VALUE function, 11

W

WHERE clause
determining whether a string is alphanumeric, 120
finding rows that satisfy multiple conditions, 2
RECURSIVE with, 537
referencing an aliased column in, 5
retrieving a subset of rows from a table, 2
ROWNUM with, 9
whole numbers, converting to binary, 474-477
wildcard (%) operator, 13
window functions, xi, 507-533
advantages, 396, 481, 530-531
NULLs and, 523
partitions, 521-523

platforms supporting, 422, 483
referencing in WHERE clause, 5
reports and, 532-533
timing of, 315, 421
WITH clause (DB2/SQL Server), 440, 445, 450
WITH clause (SQL Server), 272
WITH ROLLUP (SQL Server/MySQL), 397

Y

YEAR function (DB2/MySQL/SQL Server), 250, 326
Young, Kay, 354

Z

Zermelo, Ernst, 512

About the Authors

Anthony Molinaro is a data scientist at Johnson & Johnson. In his current role he is a manager in the Observational Health Data Analytics group within Janssen R&D. His primary areas of research are nonparametric methods, time-series analysis, and large-scale database characterization and transformation. He is a member of the open science OHDSI community. Anthony holds a BA in mathematics and an MA in applied mathematics & statistics from CUNY Hunter College. He resides in New York with his wife Georgia and his two daughters, Kiki and Connie.

Robert de Graaf graduated as an engineer and worked in the manufacturing industry after completing his studies. While working as an engineer, Robert discovered the power of statistics for solving real-world problems, and completed a master's degree in statistics in time to benefit from the data science boom. He has worked for RightShip as their senior data scientist since 2013, and is the author of *Managing Your Data Science Projects* (Apress).

Colophon

The animal on the cover of *SQL Cookbook* is a starred agama or roughtail rock agama (*Stellagama stellio*). These lizards can be found in Egypt, Turkey, Greece, and other countries surrounding the Mediterranean Sea, and are often present in rocky mountainous and coastal regions with arid or semi-arid climates. Starred agamas are diurnal and can often be found on rocks, trees, buildings, and other habitats that allow for climbing and hiding.

Starred agamas lay anywhere from 3 to 12 eggs per clutch, and they grow to about 30–35 cm in length. This species is characterized by strong legs and—like many other agamids—the ability to change color depending on their mood or the surrounding temperature. Both males and females typically have gray or brown bodies with colorful spots along their back and sides. Unlike other lizards, agamids such as the starred agama cannot regenerate their tails if they lose them.

Though they can be skittish, starred agamas are not usually aggressive toward humans and become quite tame if handled from a young age. They are commonly kept as pets, and can be fed a combination of insects and various leafy greens. Small groups of agamas can be housed together if the terrarium is large enough, but males need to be kept separate from one another to prevent fighting.

The IUCN does not list the starred agama as a species of concern, and its population is stable. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving, loose plate, source unknown. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning

©2019 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175