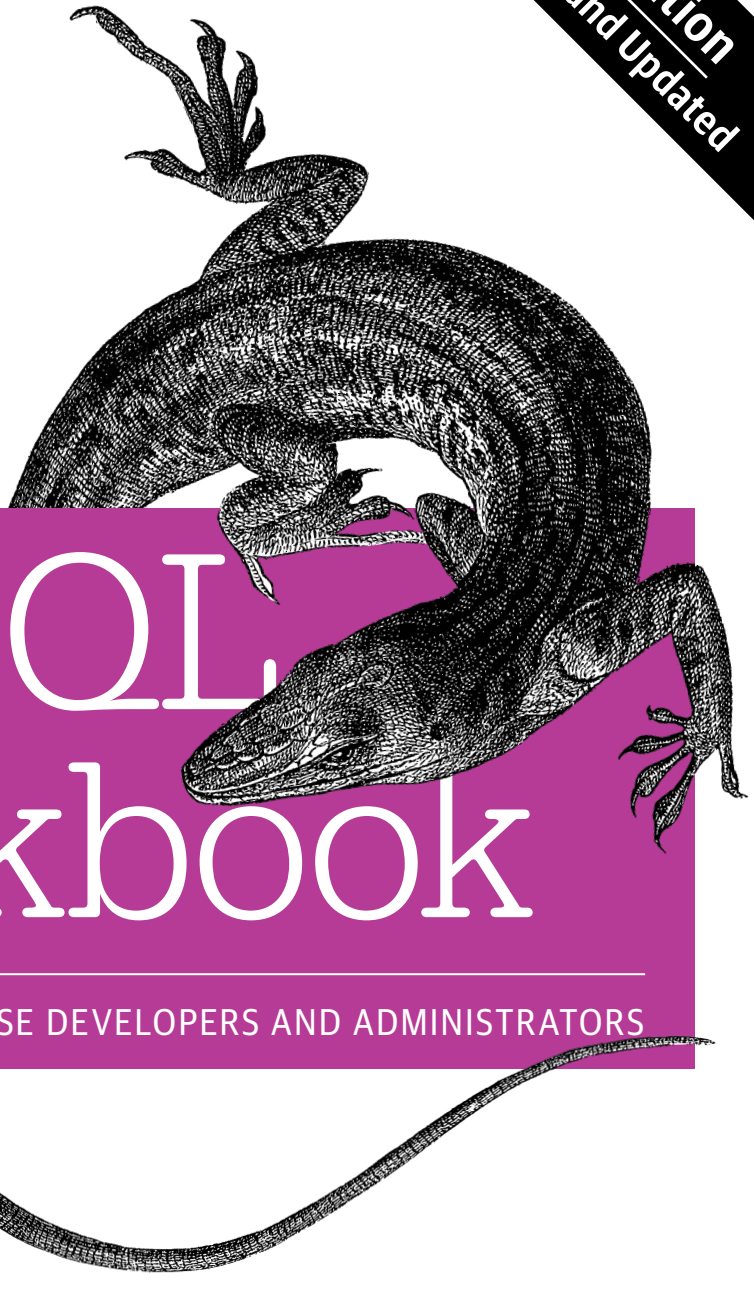
O'REILLY®

3rd Edition
Revised and Updated



MySQL Cookbook

SOLUTIONS FOR DATABASE DEVELOPERS AND ADMINISTRATORS

Paul DuBois

O'REILLY®

MySQL Cookbook

MySQL's popularity has brought a flood of questions about how to solve specific problems, and that's where this cookbook is essential. When you need quick solutions or techniques, this handy resource provides scores of short, focused pieces of code, hundreds of worked-out examples, and clear, concise explanations for programmers who don't have the time (or expertise) to solve MySQL problems from scratch.

Ideal for beginners and professional database and web developers, this updated third edition covers powerful features in MySQL 5.6 (and some in 5.7). The book focuses on programming APIs in Python, PHP, Java, Perl, and Ruby. With more than 200+ recipes, you'll learn how to:

- Use the mysql client and write MySQL-based programs
- Create, populate, and select data from tables
- Store, retrieve, and manipulate strings
- Work with dates and times
- Sort query results and generate summaries
- Use stored routines, triggers, and scheduled events
- Import, export, validate, and reformat data
- Perform transactions and work with statistics
- Process web input, and generate web content from query results
- Use MySQL-based web session management
- Provide security and server administration

"A true classic, this remains the best collection of MySQL recipes available. This book covers basics needed by beginners, and presents the very latest developments that advanced users can use to deepen their knowledge. And it's crowded with tips that make it even more valuable to MySQL professionals."

—Ulf Wendel,
Senior Software Engineer for MySQL and co-author of the mysqlnd PHP library

Paul DuBois is one of the primary contributors to the MySQL Reference Manual, a renowned online manual that has supported MySQL administrators and database developers for years. He's a member of the MySQL documentation team at Oracle and author of several books.

DATABASES

US $79.99 **CAN $83.99**
ISBN: 978-1-449-37402-0

Twitter: @oreillymedia
facebook.com/oreilly

THIRD EDITION

MySQL Cookbook

Paul DuBois

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

MySQL Cookbook , Third Edition

by Paul DuBois

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Andy Oram and Allyson MacDonald
Production Editor: Nicole Shelby
Proofreader: Kim Cofer
Indexer: Lucie Haskins

Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2002: First Edition

November 2006: Second Edition

August 2014: Third Edition

Revision History for the Third Edition:

2014-07-25: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449374020* for release details.

ISBN: 978-1-449-37402-0

[LSI]

Table of Contents

Preface

The MySQL database management system is popular for many reasons. It's fast, and it's easy to set up, use, and administer. It runs under many varieties of Unix and Windows, and MySQL-based programs can be written in many languages.

MySQL's popularity raises the need to address questions its users have about how to solve specific problems. That is the purpose of *MySQL Cookbook*: to serve as a handy resource to which you can turn for quick solutions or techniques for attacking particular types of questions that come up when you use MySQL. Naturally, because it's a cookbook, it contains recipes: straightforward instructions you can follow rather than develop your own code from scratch. It's written using a problem-and-solution format designed to be extremely practical and to make the contents easy to read and assimilate. It contains many short sections, each describing how to write a query, apply a technique, or develop a script to solve a problem of limited and specific scope. This book doesn't develop full-fledged, complex applications. Instead, it assists you in developing such applications yourself by helping you get past problems that have you stumped.

For example, a common question is, "How can I deal with quotes and special characters in data values when I'm writing queries?" That's not difficult, but figuring out how to do it is frustrating when you're not sure where to start. This book demonstrates what to do; it shows you where to begin and how to proceed from there. This knowledge will serve you repeatedly because after you see what's involved, you'll be able to apply the technique to any kind of data, such as text, images, sound or video clips, news articles, compressed files, or PDF documents. Another common question is, "Can I access data from multiple tables at the same time?" The answer is "Yes," and it's easy to do because it's just a matter of knowing the proper SQL syntax. But it's not always clear how until you see examples, which this book gives you. Other techniques that you'll learn from this book include how to:

- Use SQL to select, sort, and summarize rows
- Find matches or mismatches between tables

- Perform transactions
- Determine intervals between dates or times, including age calculations
- Identify or remove duplicate rows
- Use `LOAD DATA` to read your datafiles properly or find which values in the file are invalid
- Use strict mode to prevent entry of bad data into your database
- Generate sequence numbers to use as unique row identifiers
- Use a view as a "virtual table"
- Write stored procedures and functions, set up triggers that activate to perform specific data-handling operations when you insert or update table rows, and use the Event Scheduler to run queries on a schedule
- Generate web pages from database content
- Manage user accounts
- Control server logging

One part of using MySQL is understanding how to communicate with the server—that is, how to use SQL, the language in which queries are formulated. Therefore, one major emphasis of this book is using SQL to formulate queries that answer particular kinds of questions. One helpful tool for learning and using SQL is the *mysql* client program that is included in MySQL distributions. You can use client interactively to send SQL statements to the server and see the results. This is extremely useful because it provides a direct interface to SQL; so useful, in fact, that the first chapter is devoted to *mysql*.

But the ability to issue SQL queries alone is not enough. Information extracted from a database often requires further processing or presentation in a particular way. What if you have queries with complex interrelationships, such as when you need to use the results of one query as the basis for others? What if you need to generate a specialized report with very specific formatting requirements? These problems bring us to the other major emphasis of the book—how to write programs that interact with the MySQL server through an application programming interface (API). When you know how to use MySQL from within the context of a programming language, you gain other ways to exploit MySQL's capabilities:

- You can save query results and reuse them later.
- You have full access to the expressive power of a general-purpose programming language. This enables you to make decisions based on success or failure of a query, or on the content of the rows that are returned, and then tailor the actions taken accordingly.

- You can format and display query results however you like. If you're writing a command-line script, you can generate plain text. If it's a web-based script, you can generate an HTML table. If it's an application that extracts information for transfer to some other system, you might generate a datafile expressed in XML.

Combining SQL with a general-purpose programming language gives you an extremely flexible framework for issuing queries and processing their results. Programming languages increase your capability to perform complex database operations. But that doesn't mean this book is complex. It keeps things simple, showing how to construct small building blocks using techniques that are easy to understand and easily mastered.

I'll leave it to you to combine these techniques in your own programs, which you can do to produce arbitrarily complex applications. After all, the genetic code is based on only four nucleic acids, but these basic elements have been combined to produce the astonishing array of biological life we see all around us. Similarly, there are only 12 notes in the scale, but in the hands of skilled composers, they are interwoven to produce a rich and endless variety of music. In the same way, when you take a set of simple recipes, add your imagination, and apply them to the database programming problems you want to solve, you can produce applications that perhaps are not works of art, but are certainly useful and will help you and others be more productive.

Who This Book Is For

This book will be useful for anybody who uses MySQL, ranging from individuals who want to use a database for personal projects such as a blog or wiki, to professional database and web developers. The book is also intended for people who do not now use MySQL, but would like to. For example, it will be useful if you want to learn about databases but realize that a "big" database system such as Oracle can be daunting as a learning tool. (Perhaps I shouldn't say that. Oracle bought MySQL in 2010 and is now my employer!)

If you're new to MySQL, you'll find lots of ways to use it here that may be new to you. If you're more experienced, you're probably already familiar with many of the problems addressed here, but may not have had to solve them before and should find the book a great timesaver. Take advantage of the recipes given in the book and use them in your own programs rather than writing the code from scratch.

The material ranges from introductory to advanced, so if a recipe describes techniques that seem obvious to you, skip it. Conversely, if you don't understand a recipe, set it aside and come back to it later, perhaps after reading some of the other recipes.

What's in This Book

It's very likely when you use this book that you're trying to develop an application but are not sure how to implement certain pieces of it. In this case, you already know what type of problem you want to solve; check the table of contents or the index for a recipe that shows how to do what you want. Ideally, the recipe will be just what you had in mind. Alternatively, you may be able to adapt a recipe for a similar problem to suit the issue at hand. I explain the principles involved in developing each technique so that you can modify it to fit the particular requirements of your own applications.

Another way to approach this book is to just read through it with no specific problem in mind. This can give you a broader understanding of the things MySQL can do, so I recommend that you page through the book occasionally. It's a more effective tool if you know the kinds of problems it addresses.

As you get into later chapters, you'll find recipes that assume a knowledge of topics covered in earlier chapters. This also applies within a chapter, where later sections often use techniques discussed earlier in the chapter. If you jump into a chapter and find a recipe that uses a technique with which you're not familiar, check the table of contents or the index to find where the technique is explained earlier. For example, if a recipe sorts a query result using an `ORDER BY` clause that you don't understand, turn to Chapter 7, which discusses various sorting methods and explains how they work.

Here's a summary of each chapter to give you an overview of the book's contents.

Chapter 1, *Using the mysql Client Program*, describes how to use the standard MySQL command-line client. *mysql* is often the first or primary interface to MySQL that people use, and it's important to know how to exploit its capabilities. This program enables you to issue queries and see their results interactively, so it's good for quick experimentation. You can also use it in batch mode to execute canned SQL scripts or send its output into other programs. In addition, the chapter discusses other ways to use *mysql*, such as how to make long lines more readable or generate output in various formats.

Chapter 2, *Writing MySQL-Based Programs*, demonstrates the essential elements of MySQL programming: how to connect to the server, issue queries, retrieve the results, and handle errors. It also discusses how to handle special characters and `NULL` values in queries, how to write library files to encapsulate code for commonly used operations, and various ways to gather the parameters needed for making connections to the server.

Chapter 3, *Selecting Data from Tables*, covers several aspects of the `SELECT` statement, which is the primary vehicle for retrieving data from the MySQL server: specifying which columns and rows you want to retrieve, dealing with `NULL` values, and selecting one section of a query result. Later chapters cover some of these topics in more detail, but this chapter provides an overview of the concepts on which they depend if you need some introductory background on row selection or don't yet know a lot about SQL.

Chapter 4, *Table Management*, covers table cloning, copying results into other tables, using temporary tables, and checking or changing a table's storage engine.

Chapter 5, *Working with Strings*, describes how to deal with string data. It covers character sets and collations, string comparisons, dealing with case-sensitivity issues, pattern matching, breaking apart and combining strings, and performing `FULLTEXT` searches.

Chapter 6, *Working with Dates and Times*, shows how to work with temporal data. It describes MySQL's date format and how to display date values in other formats. It also covers how to use MySQL's special `TIMESTAMP` data type, how to set the time zone, how to convert between different temporal units, how to perform date arithmetic to compute intervals or generate one date from another, and how to perform leap-year calculations.

Chapter 7, *Sorting Query Results*, describes how to put the rows of a query result in the order you want. This includes specifying the sort direction, dealing with `NULL` values, accounting for string case sensitivity, and sorting by dates or partial column values. It also provides examples that show how to sort special kinds of values, such as domain names, IP numbers, and `ENUM` values.

Chapter 8, *Generating Summaries*, shows techniques for assessing the general characteristics of a set of data, such as how many values it contains or its minimum, maximum, and average values.

Chapter 9, *Using Stored Routines, Triggers, and Scheduled Events*, describes how to write stored functions and procedures that are stored on the server side, triggers that activate when tables are modified, and events that execute on a scheduled basis.

Chapter 10, *Working with Metadata*, discusses how to get information *about* the data that a query returns, such as the number of rows or columns in the result, or the name and data type of each column. It also shows how to ask MySQL what databases and tables are available or determine the structure of a table.

Chapter 11, *Importing and Exporting Data*, describes how to transfer information between MySQL and other programs. This includes how to use `LOAD DATA`, convert files from one format to another, and determine table structure appropriate for a dataset.

Chapter 12, *Validating and Reformatting Data*, describes how to extract or rearrange columns in datafiles, check and validate data, and rewrite values such as dates that often come in a variety of formats.

Chapter 13, *Generating and Using Sequences*, discusses `AUTO_INCREMENT` columns, MySQL's mechanism for producing sequence numbers. It shows how to generate new sequence values or determine the most recent value, how to resequence a column, and how to use sequences to generate counters. It also shows how to use `AUTO_INCREMENT` values to maintain a master-detail relationship between tables, including pitfalls to avoid.

Chapter 14, *Using Joins and Subqueries*, shows how to perform operations that select rows from multiple tables. It demonstrates how to compare tables to find matches or mismatches, produce master-detail lists and summaries, and enumerate many-to-many relationships.

Chapter 15, *Statistical Techniques*, illustrates how to produce descriptive statistics, frequency distributions, regressions, and correlations. It also covers how to randomize a set of rows or pick rows at random from the set.

Chapter 16, *Handling Duplicates*, discusses how to identify, count, and remove duplicate rows—and how to prevent them from occurring in the first place.

Chapter 17, *Performing Transactions*, shows how to handle multiple SQL statements that must execute together as a unit. It discusses how to control MySQL's auto-commit mode and how to commit or roll back transactions.

Chapter 18, *Introduction to MySQL on the Web*, gets you set up to write web-based MySQL scripts. Web programming enables you to generate dynamic pages from database content or collect information for storage in your database. The chapter discusses how to configure Apache to run Perl, Ruby, PHP, and Python scripts, and how to configure Tomcat to run Java scripts written using JSP notation.

Chapter 19, *Generating Web Content from Query Results*, shows how to use the query results to generate various HTML structures such as paragraphs, lists, tables, hyperlinks, and navigation indexes. It also describes how to store images into MySQL and retrieve and display them later, and how to generate downloadable result sets.

Chapter 20, *Processing Web Input with MySQL*, discusses how to obtain input from users over the Web and use it to create new database rows or as the basis for performing searches. It deals heavily with form processing, including how to construct form elements such as radio buttons, pop-up menus, or checkboxes, based on information contained in your database.

Chapter 21, *Using MySQL-Based Web Session Management*, describes how to write web applications that remember information across multiple requests, using MySQL for backing store. This is useful for collecting information in stages, or when you need to make decisions based on prior user actions.

Chapter 22, *Server Administration*, is written for database administrators. It covers server configuration, the plug-in interface, log management, server monitoring, and making backups.

Chapter 23, *Security*, is another administrative chapter. It discusses user account management, including creating accounts, setting passwords, and assigning privileges. It also describes how to implement password policy, find and fix insecure accounts, and expire or unexpire passwords.

MySQL APIs Used in This Book

MySQL programming interfaces exist for many languages, including C, C++, Eiffel, Go, Java, Perl, PHP, Python, Ruby, and Tcl. Given this fact, writing a MySQL cookbook presents an author with a challenge. The book should provide recipes for doing many interesting and useful things with MySQL, but which API or APIs should the book use? Showing an implementation of every recipe in every language results either in covering very few recipes or in a very, very large book! It also results in redundancies when implementations in different languages bear a strong resemblance to each other. On the other hand, it's worthwhile taking advantage of multiple languages, because one often is more suitable than another for solving a particular problem.

To resolve this dilemma, I've chosen a small number of APIs to write the recipes in this book. This makes its scope manageable while permitting latitude to choose from multiple APIs:

- The Perl and Ruby DBI modules
- PHP, using the PDO extension
- Python, using the MySQL Connector/Python driver for the DB API
- Java, using the MySQL Connector/J driver for the JDBC interface

Why these languages? Perl and PHP were easy to pick. Perl is a widely used language that became so based on certain strengths such as its text-processing capabilities. In addition, it's very popular for writing MySQL programs. Ruby has an easy-to-use database-access module modeled after the Perl module. PHP is widely deployed, especially on the Web. One of PHP's strengths is the ease with which you can use it to access databases, making it a natural choice for MySQL scripting. Python and Java are perhaps not as popular as Perl or PHP for MySQL programming, but each has a significant number of followers. In the Java community in particular, MySQL has a strong following among developers who use JavaServer Pages (JSP) technology to build database-backed web applications.

I believe these languages taken together reflect pretty well the majority of the existing user base of MySQL programmers. If you prefer some language not shown here, be sure to pay careful attention to Chapter 2, to familiarize yourself with the book's primary APIs. Knowing how to perform database operations with the programming interfaces used here will help you translate recipes for other languages.

Version and Platform Notes

Development of the code in this book took place under MySQL 5.5, 5.6, and 5.7. Because new features are added to MySQL on a regular basis, some examples will not work under older versions. For example, MySQL 5.5 introduces authentication plug-ins, and

MySQL 5.6 introduces `TIMESTAMP`-like auto-initialization and auto-update properties for the `DATETIME` data type.

I do not assume that you are using Unix, although that is my own preferred development platform. (In this book, "Unix" also refers to Unix-like systems such as Linux and Mac OS X.) Most of the material here is applicable both to Unix and Windows.

Conventions Used in This Book

This book uses the following font conventions:

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Used to indicate text that you type when running commands.

`Constant width italic`
: Used to indicate variable input; you should substitute a value of your own choosing.

Italic
: Used for URLs, hostnames, names of directories and files, Unix commands and options, programs, and occasionally for emphasis.

This element signifies a tip or suggestion.

This element indicates a warning or caution.

This element signifies a general note.

Commands often are shown with a prompt to illustrate the context in which they are used. Commands issued from the command line are shown with a `%` prompt:

```
% chmod 600 my.cnf
```

That prompt is one that Unix users are used to seeing, but it doesn't necessarily signify that a command works only under Unix. Unless indicated otherwise, commands shown with a `%` prompt generally should work under Windows, too.

If you should run a command under Unix as the `root` user, the prompt is `#` instead:

```
# perl -MCPAN -e shell
```

Commands that are specific to Windows use the `C:\>` prompt:

```
C:\> "C:\Program Files\MySQL\MySQL Server 5.6\bin\mysql"
```

SQL statements that are issued from within the *mysql* client program are shown with a `mysql>` prompt and terminated with a semicolon:

```
mysql> SELECT * FROM my_table;
```

For examples that show a query result as you would see it when using *mysql*, I sometimes truncate the output, using an ellipsis (`...`) to indicate that the result consists of more rows than are shown. The following query produces many rows of output, from which those in the middle have been omitted:

```
mysql> SELECT name, abbrev FROM states ORDER BY name;
+----------------+--------+
| name           | abbrev |
+----------------+--------+
| Alabama        | AL     |
| Alaska         | AK     |
| Arizona        | AZ     |
…
| West Virginia  | WV     |
| Wisconsin      | WI     |
| Wyoming        | WY     |
+----------------+--------+
```

Examples that show only the syntax for SQL statements do not include the `mysql>` prompt, but they do include semicolons as necessary to make it clearer where statements end. For example, this is a single statement:

```
CREATE TABLE t1 (i INT)
SELECT * FROM t2;
```

But this example represents two statements:

```
CREATE TABLE t1 (i INT);
SELECT * FROM t2;
```

The semicolon is a notational convenience used within *mysql* as a statement terminator. But it is not part of SQL itself, so when you issue SQL statements from within programs that you write (for example, using Perl or Java), don't include terminating semicolons.

The MySQL Cookbook Companion Website

MySQL Cookbook has a companion website where you can obtain source code and sample data for examples developed throughout this book, errata, and auxiliary documentation.

The website also makes examples from the book available online so you can try them from your browser.

Recipe Source Code and Data

The examples in this book are based on source code and sample data from two distributions named `recipes` and `mcb-kjv` available at the companion website.

The `recipes` distribution is the primary source of examples, and references to it occur throughout the book. The distribution is available as a compressed TAR file (*recipes.tar.gz*) or as a ZIP file (*recipes.zip*). Either distribution format when unpacked creates a directory named *recipes*.

Use the `recipes` distribution to save yourself a lot of typing. For example, when you see a `CREATE TABLE` statement in the book that describes what a database table looks like, you'll usually find an SQL batch file in the *tables* directory that you can use to create the table instead of entering the definition manually. Change location into the *tables* directory and execute the following command, where *`filename`* is the name of the file containing the `CREATE TABLE` statement:

```
% mysql cookbook < filename
```

If you need to specify MySQL username or password options, add them to the command line.

The `recipes` distribution contains programs as shown in the book, but in many cases also includes implementations in additional languages. For example, a script shown in the book using Python may be available in the `recipes` distribution in Perl, Ruby, PHP, or Java as well. This may save you translation effort should you wish to convert a program shown in the book to a different language.

The other distribution is named `mcb-kjv` and contains the text of the King James Version of the Bible, formatted suitably for loading into MySQL. It's used in Chapter 5 as the source of a reasonably large body of text for examples that demonstrate `FULLTEXT` searches, and occasionally elsewhere in the book. This distribution is provided separately from the `recipes` distribution due to its size. It's available as a compressed TAR file (*mcb-kjv.tar.gz*) or as a ZIP file (*mcb-kjv.zip*). Either distribution format when unpacked creates a directory named *mcb-kjv*.

The `mcb-kjv` distribution is derived from KJV text originally found on the Unbound Bible site, restructured to be more usable for examples in the book. The distribution includes notes that describe the modifications I made.

MySQL Cookbook Companion Documents

Some appendixes included in previous *MySQL Cookbook* editions are now available in standalone form at the companion website. They provide background information for topics covered in the book.

- "Executing Programs from the Command Line" provides instructions for executing commands at the command prompt and setting environment variables such as `PATH`.
- "JSP, JSTL, and Tomcat Primer" provides a general overview of JavaServer Pages (JSP) programming and installation instructions for the Tomcat web server. Read this document if you need to install Tomcat or are not familiar with it, or if you've never written pages using JSP notation. It also provides an overview of the Java Standard Tag Library (JSTL) that is used heavily for JSP pages in this book. This material is background for topics covered in the web programming chapters, beginning with Chapter 18.

Obtaining MySQL and Related Software

To run the examples in this book, you need access to MySQL, as well as the appropriate MySQL-specific interfaces for the programming languages that you want to use. The following notes describe what software is required and where to get it.

If you access a MySQL server run by somebody else, you need only the MySQL client software on your own machine. To run your own server, you need a full MySQL distribution.

To write your own MySQL-based programs, you communicate with the server through a language-specific API. The Perl and Ruby interfaces rely on the MySQL C API client library to handle the low-level client-server protocol. This is also true for the PHP interface, unless PHP is configured to use `mysqlnd`, the native protocol driver. For Perl and Ruby, you must install the C client library and header files first. PHP includes the required MySQL client support files, but must be compiled with MySQL support enabled or you won't be able to use it. The Python and Java drivers for MySQL implement the client-server protocol directly, so they do not require the MySQL C client library.

You may not need to install the client software yourself—it might already be present on your system. This is a common situation if you have an account with an Internet service

provider (ISP) that provides services such as a web server already enabled for access to MySQL.

MySQL

MySQL distributions and documentation, including the *MySQL Reference Manual,* are available from *http://dev.mysql.com/downloads* and *http://dev.mysql.com/doc*.

If you need to install the MySQL C client library and header files, they're included when you install MySQL from a source distribution, or when you install MySQL using a binary (precompiled) distribution other than an RPM binary distribution. Under Linux, you have the option of installing MySQL using RPM files, but the client library and header files are not installed unless you install the development RPM. (There are separate RPM files for the server, the standard client programs, and the development libraries and header files.) If you don't install the development RPM, you'll join the many Linux users who've asked, "I installed MySQL, but I cannot find the libraries or header files; where are they?"

Perl Support

General Perl information is available on the Perl Programming Language website.

You can obtain Perl software from the Comprehensive Perl Archive Network (CPAN).

To write MySQL-based Perl programs, you need the DBI module and the MySQL-specific DBD module, DBD::mysql.

To install these modules under Unix, let Perl itself help you. For example, to install DBI and DBD::mysql, run the following commands (you'll probably need to do this as `root`):

```
# perl -MCPAN -e shell
cpan> install DBI
cpan> install DBD::mysql
```

If the last command complains about failed tests, use `force install DBD::mysql` instead. Under ActiveState Perl for Windows, use the *ppm* utility:

```
C:\> ppm
ppm> install DBI
ppm> install DBD-mysql
```

You can also use the CPAN shell or *ppm* to install other Perl modules mentioned in this book.

Once the DBI and DBD::mysql modules are installed, documentation is available from the command line:

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

Documentation is also available from the Perl website.

Ruby Support

The primary Ruby website provides access to Ruby distributions and documentation.

The Ruby DBI and MySQL driver modules are available from RubyGems; the Ruby DBI driver for MySQL requires the `mysql-ruby` module, also available from RubyGems.

To use session support as described in Chapter 21, you need the `mysql-session` package. It's available from the *MySQL Cookbook* companion website described earlier in this Preface. Obtain the `mysql-session` package, unpack it, and install its *mysqlstore.rb* and *sqlthrow.rb* files in some directory that your Ruby interpreter searches when looking for library files (see Recipe 2.3).

PHP Support

The primary PHP website provides access to PHP distributions and documentation, including PDO documentation.

PHP source distributions include PDO support, so you need not obtain it separately. However, you must enable PDO support for MySQL when you configure the distribution. If you use a binary distribution, be sure that it includes PDO MySQL support.

Python Support

The primary Python website provides access to Python distributions and documentation. General documentation for the DB API database access interface is on the Python Wiki.

For MySQL Connector/Python, the driver module that provides MySQL connectivity for the DB API, distributions and documentation are available from *http://bit.ly/py-connect* and *http://bit.ly/py-dev-guide*.

Java Support

You need a Java compiler to build and run Java programs. The *javac* and *jikes* compilers are two possible choices. On many systems, you'll find one or both installed already. Otherwise, you can get a compiler as part of the Java Development Kit (JDK). If no JDK is installed on your system, versions are available for Solaris, Linux, and Windows at Oracle's Java site. The same site provides access to documentation (including the specifications) for JDBC, servlets, JavaServer Pages (JSP), and the JSP Standard Tag Library (JSTL).

For MySQL Connector/J, the driver that provides MySQL connectivity for the JDBC interface, distributions and documentation are available from *http://bit.ly/jconn-dl* and *http://bit.ly/j-dev-guide*.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*MySQL Cookbook, Third Edition* by Paul DuBois (O'Reilly). Copyright 2014 Paul DuBois, 978-1-449-37402-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

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 *http://bit.ly/mysql_ckbk_3e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.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*

Acknowledgments

To each reader, thank you for reading my book. I hope that it serves you well and that you find it useful.

Thanks to my technical reviewers, Johannes Schlüter, Geert Vanderkelen, and Ulf Wendel. They made several corrections and suggestions that improved the text in many ways, and I appreciate their help.

Andy Oram prodded me to begin the third edition and served as its editor, Nicole Shelby guided the book through production, and Kim Cofer and Lucie Haskins provided proofreading and indexing.

Thanks to my wife Karen, whose encouragement and support throughout the writing process means more than I can say.

CHAPTER 1

Using the mysql Client Program

1.0. Introduction

The MySQL database system uses a client-server architecture. The server, *mysqld*, is the program that actually manipulates databases. To tell the server what to do, use a client program that communicates your intent by means of statements written in Structured Query Language (SQL). Client programs are written for diverse purposes, but each interacts with the server by connecting to it, sending SQL statements to have database operations performed, and receiving the results.

Clients are installed locally on the machine from which you want to access MySQL, but the server can be installed anywhere, as long as clients can connect to it. Because MySQL is an inherently networked database system, clients can communicate with a server running locally on your own machine or somewhere on the other side of the planet.

The *mysql* program is one of the clients included in MySQL distributions. When used interactively, *mysql* prompts you for a statement, sends it to the MySQL server for execution, and displays the results. *mysql* also can be used noninteractively in batch mode to read statements stored in files or produced by programs. This enables use of *mysql* from within scripts or *cron* jobs, or in conjunction with other applications.

This chapter describes *mysql*'s capabilities so that you can use it more effectively:

- Setting up a MySQL account for using the `cookbook` database
- Specifying connection parameters and using option files
- Executing SQL statements interactively and in batch mode
- Controlling *mysql* output format
- Using user-defined variables to save information

To try for yourself the examples shown in this book, you need a MySQL user account and a database. The first two recipes in this chapter describe how to use *mysql* to set those up, based on these assumptions:

- The MySQL server is running locally on your own system
- Your MySQL username and password are `cbuser` and `cbpass`
- Your database is named `cookbook`

If you like, you can violate any of the assumptions. Your server need not be running locally, and you need not use the username, password, or database name that are used in this book. Naturally, in such cases, you must modify the examples accordingly.

Even if you choose not to use `cookbook` as your database name, I recommend that you use a database dedicated to the examples shown here, not one that you also use for other purposes. Otherwise, the names of existing tables may conflict with those used in the examples, and you'll have to make modifications that would be unnecessary with a dedicated database.

Scripts that create the tables used in this chapter are located in the *tables* directory of the `recipes` distribution that accompanies *MySQL Cookbook*. Other scripts are located in the *mysql* directory. To get the `recipes` distribution, see the Preface.

Alternatives to the mysql Program

The *mysql* client is not the only program you can use for executing queries. For example, you might prefer the graphical MySQL Workbench program, which provides a point-and-click interface to MySQL servers. Another popular interface is phpMyAdmin, which enables you to access MySQL through your web browser. If you execute queries other than by using *mysql*, some concepts covered in this chapter may not apply.

1.1. Setting Up a MySQL User Account

Problem

You need an account for connecting to your MySQL server.

Solution

Use `CREATE USER` and `GRANT` statements to set up the account. Then use the account name and password to make connections to the server.

Discussion

Connecting to a MySQL server requires a username and password. You may also need to specify the name of the host on which the server is running. If you don't specify connection parameters explicitly, *mysql* assumes default values. For example, given no explicit hostname, *mysql* assumes that the server is running on the local host.

If someone else has already set up an account for you, just use that account. Otherwise, the following example shows how to use the *mysql* program to connect to the server and issue the statements that set up a user account with privileges for accessing a database named `cookbook`. The arguments to *mysql* include `-h localhost` to connect to the MySQL server running on the local host, `-u root` to connect as the MySQL `root` user, and `-p` to tell *mysql* to prompt for a password:

```
% mysql -h localhost -u root -p
Enter password: ******
mysql> CREATE USER 'cbuser'@'localhost' IDENTIFIED BY 'cbpass';
mysql> GRANT ALL ON cookbook.* TO 'cbuser'@'localhost';
Query OK, 0 rows affected (0.09 sec)
mysql> quit
Bye
```

If when you attempt to invoke *mysql* the result is an error message that it cannot be found or is an invalid command, that means your command interpreter doesn't know where *mysql* is installed. See Recipe 1.3 for information about setting the `PATH` environment variable that the interpreter uses to find commands.

In the commands shown, the `%` represents the prompt displayed by your shell or command interpreter, and `mysql>` is the prompt displayed by *mysql*. Text that you type is shown in bold. Nonbold text (including the prompts) is program output; don't type any of that.

When *mysql* prints the password prompt, enter the MySQL `root` password where you see the `******`; if the MySQL `root` user has no password, just press the Enter (or Return) key at the password prompt. Then enter the `CREATE USER` and `GRANT` statements as shown.

The `quit` command terminates your *mysql* session. You can also terminate a session by using an `exit` command or (under Unix) by typing Ctrl-D.

To grant the `cbuser` account access to a database other than `cookbook`, substitute the database name where you see `cookbook` in the `GRANT` statement. To grant access for the `cookbook` database to an existing account, omit the `CREATE USER` statement and substitute that account for `'cbuser'@'localhost'` in the `GRANT` statement.

The hostname part of `'cbuser'@'localhost'` indicates the host *from which* you'll connect to the MySQL server. To set up an account that will connect to a server running on the local host, use `localhost`, as shown. If you plan to connect to the server from another host, substitute that host in the `CREATE USER` and `GRANT` statements. For example, if you'll

connect to the server from a host named *myhost.example.com*, the statements look like this:

```
mysql> CREATE USER 'cbuser'@'myhost.example.com' IDENTIFIED BY 'cbpass';
mysql> GRANT ALL ON cookbook.* TO 'cbuser'@'myhost.example.com';
```

It may have occurred to you that there's a paradox in the procedure just described: to set up a `cbuser` account that can connect to the MySQL server, you must first connect to the server so that you can execute the `CREATE USER` and `GRANT` statements. I'm assuming that you can already connect as the MySQL `root` user because `CREATE USER` and `GRANT` can be used only by a user such as `root` that has the administrative privileges needed to set up other user accounts. If you can't connect to the server as `root`, ask your MySQL administrator to create the `cbuser` account for you.

MySQL Accounts and Login Accounts

MySQL accounts differ from login accounts for your operating system. For example, the MySQL `root` user and the Unix `root` user are separate and have nothing to do with each other, even though the username is the same in each case. This means they very likely have different passwords. It also means you don't create new MySQL accounts by creating login accounts for your operating system; use `CREATE USER` and `GRANT` instead.

After creating the `cbuser` account, verify that you can use it to connect to the MySQL server. From the host that was named in the `CREATE USER` statement, run the following command to do this (the host named after `-h` should be the host where the MySQL server is running):

```
% mysql -h localhost -u cbuser -p
Enter password: cbpass
```

Now you can proceed to create the `cookbook` database and tables within it, as described in Recipe 1.2. To make it easier to invoke *mysql* without specifying connection parameters each time, put them in an option file (see Recipe 1.4).

See Also

For additional information about administering MySQL accounts, see Chapter 23.

1.2. Creating a Database and a Sample Table

Problem

You want to create a database and set up tables within it.

Solution

Use a `CREATE DATABASE` statement to create the database, a `CREATE TABLE` statement for each table, and `INSERT` statements to add rows to the tables.

Discussion

The `GRANT` statement shown in Recipe 1.1 sets up privileges for accessing the `cookbook` database but does not create the database. This section shows how to do that, and also how to create a table and load it with the sample data used for examples in the following sections. Similar instructions apply for creating other tables used elsewhere in this book.

Connect to the MySQL server as shown at the end of Recipe 1.1, then create the database like this:

```
mysql> CREATE DATABASE cookbook;
```

Now that you have a database, you can create tables in it. First, select `cookbook` as the default database:

```
mysql> USE cookbook;
```

Then create a simple table:

```
mysql> CREATE TABLE limbs (thing VARCHAR(20), legs INT, arms INT);
```

And populate it with a few rows:

```
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('human',2,2);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('insect',6,0);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('squid',0,10);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('fish',0,0);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('centipede',100,0);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('table',4,0);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('armchair',4,2);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('phonograph',0,1);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('tripod',3,0);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('Peg Leg Pete',1,2);
mysql> INSERT INTO limbs (thing,legs,arms) VALUES('space alien',NULL,NULL);
```

Here's a tip for entering the `INSERT` statements more easily: after entering the first one, press the up arrow to recall it, press Backspace (or Delete) a few times to erase characters back to the last open parenthesis, then type the data values for the next statement. Or, to avoid typing the `INSERT` statements altogether, skip ahead to Recipe 1.6.

The table you just created is named `limbs` and contains three columns to record the number of legs and arms possessed by various life forms and objects. The physiology of the alien in the last row is such that the proper values for the `arms` and `legs` columns cannot be determined; `NULL` indicates "unknown value."

Verify that the rows were added to the `limbs` table by executing a `SELECT` statement:

```
mysql> SELECT * FROM limbs;
+--------------+------+------+
| thing        | legs | arms |
+--------------+------+------+
| human        |    2 |    2 |
| insect       |    6 |    0 |
| squid        |    0 |   10 |
| fish         |    0 |    0 |
| centipede    |  100 |    0 |
| table        |    4 |    0 |
| armchair     |    4 |    2 |
| phonograph   |    0 |    1 |
| tripod       |    3 |    0 |
| Peg Leg Pete |    1 |    2 |
| space alien  | NULL | NULL |
+--------------+------+------+
```

At this point, you're all set up with a database and a table. For additional information about executing SQL statements, see Recipes 1.5 and 1.6.

In this book, statements show SQL keywords such as `SELECT` or `INSERT` in uppercase for distinctiveness. That's only a typographical convention; keywords can be any lettercase.

1.3. What to Do if mysql Cannot Be Found

Problem

When you invoke *mysql* from the command line, your command interpreter can't find it.

Solution

Add the directory where *mysql* is installed to your `PATH` setting. Then you can run *mysql* from any directory easily.

Discussion

If your shell or command interpreter can't find *mysql* when you invoke it, you'll see some sort of error message. It might look like this under Unix:

```
% mysql
mysql: Command not found.
```

Or like this under Windows:

```
C:\> mysql
Bad command or invalid filename
```

One way to tell your command interpreter where to find *mysql* is to type its full pathname each time you run it. The command might look like this under Unix:

```
% /usr/local/mysql/bin/mysql
```

Or like this under Windows:

```
C:\> "C:\Program Files\MySQL\MySQL Server 5.6\bin\mysql"
```

Typing long pathnames gets tiresome pretty quickly. You can avoid doing so by changing location into the directory where *mysql* is installed before you run it. But if you do that, you may be tempted to put all your datafiles and SQL batch files in the same directory as *mysql*, thus unnecessarily cluttering up a location intended only for programs.

A better solution is to modify your `PATH` search-path environment variable, which specifies directories where the command interpreter looks for commands. Add to the `PATH` value the directory where *mysql* is installed. Then you can invoke *mysql* from any location by entering only its name, which eliminates pathname typing. For instructions on setting your `PATH` variable, read "Executing Programs from the Command Line" on the companion website (see the Preface).

A significant additional benefit of being able to easily run *mysql* from anywhere is that you need not put your datafiles in the directory where *mysql* is located. You're free to organize your files in a way that makes sense to you, not a way imposed by some artificial necessity. For example, you can create a directory under your home directory for each database you have and put the work files associated with a given database in the appropriate directory. (I point out the importance of `PATH` here because many newcomers to MySQL aren't aware of the existence of such a thing, and consequently try to do all their MySQL-related work in the *bin* directory where *mysql* is installed.)

On Windows, another way to avoid typing the pathname or changing into the *mysql* directory is to create a shortcut and place it in a more convenient location such as the desktop. This makes it easy to start *mysql* simply by opening the shortcut. To specify command options or the startup directory, edit the shortcut's properties. If you don't always invoke *mysql* with the same options, it might be useful to create one shortcut for each set of options you need. For example, create one shortcut to connect as an ordinary user for general work and another to connect as the MySQL `root` user for administrative purposes.

1.4. Specifying mysql Command Options

Problem

When you invoke the *mysql* program without command options, it exits immediately with an "access denied" message.

Solution

You must specify connection parameters. Do this on the command line, in an option file, or using a mix of the two.

Discussion

If you invoke *mysql* with no command options, the result may be an "access denied" error. To avoid that, connect to the MySQL server as shown in Recipe 1.1, using *mysql* like this:

```
% mysql -h localhost -u cbuser -p
Enter password: cbpass
```

Each option is the single-dash "short" form: `-h` and `-u` to specify the hostname and username, and `-p` to be prompted for the password. There are also corresponding double-dash "long" forms: `--host`, `--user`, and `--password`. Use them like this:

```
% mysql --host=localhost --user=cbuser --password
Enter password: cbpass
```

To see all options that *mysql* supports, use this command:

```
% mysql --help
```

The way you specify command options for *mysql* also applies to other MySQL programs such as *mysqldump* and *mysqladmin*. For example, to generate a dump file named *cookbook.sql* that contains a backup of the tables in the `cookbook` database, execute *mysqldump* like this:

```
% mysqldump -h localhost -u cbuser -p cookbook > cookbook.sql
Enter password: cbpass
```

Some operations require an administrative MySQL account. The *mysqladmin* program can perform operations that are available only to the MySQL `root` account. For example, to stop the server, invoke *mysqladmin* as follows:

```
% mysqladmin -h localhost -u root -p shutdown
Enter password:          ← enter MySQL root account password here
```

If the value that you use for an option is the same as its default value, you can omit the option. However, there is no default password. If you like, you can specify the password directly on the command line by using `-p`*`password`* (with *no space* between the option

and the password) or `--password=`*`password`*. I don't recommend this because the password is visible to onlookers and, on multiple-user systems, may be discoverable to other users who run tools such as *ps* that report process information.

Because the default host is `localhost`, the same value we've been specifying explicitly, you can omit the `-h` (or `--host`) option from the command line:

```
% mysql -u cbuser -p
```

But suppose that you'd really rather not specify *any* options. How can you get *mysql* to "just know" what values to use? That's easy because MySQL programs support option files:

- If you put an option in an option file, you need not specify it on the command line each time you invoke a given program.
- You can mix command-line and option-file options. This enables you to store the most commonly used option values in a file but override them as desired on the command line.

The rest of this section describes these capabilities.

The Meaning of localhost in MySQL

One of the parameters you specify when connecting to a MySQL server is the host where the server is running. Most programs treat the hostname *localhost* and the IP address `127.0.0.1` as synonyms for "the local host." Under Unix, MySQL programs behave differently: by convention, they treat the hostname *localhost* specially and attempt to connect to the local server using a Unix domain socket file. To force a TCP/IP connection to the local server, use the IP address `127.0.0.1` (or `::1` if your system is configured to support IPv6) rather than the hostname *localhost*. Alternatively, you can specify a `--protocol=tcp` option to force use of TCP/IP for connecting.

The default port number is 3306 for TCP/IP connections. The pathname for the Unix domain socket varies, although it's often */tmp/mysql.sock*. To name the socket file pathname explicitly, use `-S` *`file_name`* or `--socket=`*`file_name`*.

Specifying connection parameters using option files

To avoid entering options on the command line each time you invoke *mysql*, put them in an option file for *mysql* to read automatically. Option files are plain-text files:

- Under Unix, your personal option file is named *.my.cnf* in your home directory. There are also site-wide option files that administrators can use to specify param-

eters that apply globally to all users. You can use the *my.cnf* file in the */etc* or */etc/mysql* directory, or in the *etc* directory under the MySQL installation directory.

- Under Windows, files you can use include the *my.ini* or *my.cnf* file in your MySQL installation directory (for example, *C:\Program Files\MySQL\MySQL Server 5.6*), your Windows directory (likely *C:\WINDOWS*), or the *C:* directory.

To see the exact list of permitted option-file locations, invoke *mysql* `--help`.

The following example illustrates the format used in MySQL option files:

```
# general client program connection options
[client]
host     = localhost
user     = cbuser
password = cbpass

# options specific to the mysql program
[mysql]
skip-auto-rehash
pager="/usr/bin/less -E" # specify pager for interactive mode
```

With connection parameters listed in the `[client]` group as just shown, you can connect as `cbuser` by invoking *mysql* with no options on the command line:

```
% mysql
```

The same holds for other MySQL client programs, such as *mysqldump*.

MySQL option files have these characteristics:

- Lines are written in groups (or sections). The first line of a group specifies the group name within square brackets, and the remaining lines specify options associated with the group. The example file just shown has a `[client]` group and a `[mysql]` group. To specify options for the server, *mysqld*, put them in a `[mysqld]` group.
- The usual option group for specifying client connection parameters is `[client]`. This group actually is used by all the standard MySQL clients. By listing an option in this group, you make it easier to invoke not only *mysql*, but also other programs such as *mysqldump* and *mysqladmin*. Just make sure that any option you put in this group is understood by *all* client programs. Otherwise, invoking any client that does not understand it results in an "unknown option" error.
- You can define multiple groups in an option file. By convention, MySQL clients look for parameters in the `[client]` group and in the group named for the program itself. This provides a convenient way to list general client parameters that you want all client programs to use, but you can still specify options that apply only to a particular program. The preceding sample option file illustrates this convention for the *mysql* program, which gets general connection parameters from the `[client]`

group and also picks up the `skip-auto-rehash` and `pager` options from the `[mysql]` group.

- Within a group, write option lines in *`name=value`* format, where *`name`* corresponds to an option name (without leading dashes) and *`value`* is the option's value. If an option takes no value (such as `skip-auto-rehash`), list the name by itself with no trailing *`=value`* part.
- In option files, only the long form of an option is permitted, not the short form. For example, on the command line, the hostname can be given using either `-h` *`host_name`* or `--host=`*`host_name`*. In an option file, only `host=`*`host_name`* is permitted.
- Many programs, *mysql* and *mysqld* included, have program variables in addition to command options. (For the server, these are called system variables; see Recipe 22.1.) Program variables can be specified in option files, just like options. Internally, program variable names use underscores, but in option files, you can write options and variables using dashes or underscores interchangeably. For example, `skip-auto-rehash` and `skip_auto_rehash` are equivalent. To set the server's `sql_mode` system variable in a `[mysqld]` option group, `sql_mode=`*`value`* and `sql-mode=`*`value`* are equivalent. (Interchangeability of dash and underscore also applies for options or variables specified on the command line.)
- In option files, spaces are permitted around the `=` that separates an option name and value. This contrasts with command lines, where no spaces around `=` are permitted.
- If an option value contains spaces or other special characters, you can quote it using single or double quotes. The `pager` option illustrates this.
- It's common to use an option file to specify options for connection parameters (such as `host`, `user`, and `password`). However, the file can list options that have other purposes. The `pager` option shown for the `[mysql]` group specifies the paging program that *mysql* should use for displaying output in interactive mode. It has nothing to do with how the program connects to the server.
- If a parameter appears multiple times in an option file, the last value found takes precedence. Normally, you should list any program-specific groups following the `[client]` group so that if there is any overlap in the options set by the two groups, the more general options are overridden by the program-specific values.
- Lines beginning with `#` or `;` characters are ignored as comments. Blank lines are ignored, too. `#` can be used to write comments at the end of option lines, as shown for the `pager` option.
- Options that specify file or directory pathnames should be written using `/` as the pathname separator character, even under Windows, which uses `\` as the pathname

separator. Alternatively, write \ by doubling it as \\ (this is necessary because \ is the MySQL escape character in strings).

To find out which options the *mysql* program will read from option files, use this command:

```
% mysql --print-defaults
```

You can also use the *my_print_defaults* utility, which takes as arguments the names of the option-file groups that it should read. For example, *mysqldump* looks in both the `[client]` and `[mysqldump]` groups for options. To check which option-file settings are in those groups, use this command:

```
% my_print_defaults client mysqldump
```

Mixing command-line and option-file parameters

It's possible to mix command-line options and options in option files. Perhaps you want to list your username and server host in an option file, but would rather not store your password there. That's okay; MySQL programs first read your option file to see what connection parameters are listed there, then check the command line for additional parameters. This means you can specify some options one way, and some the other way. For example, you can list your username and hostname in an option file, but use a password option on the command line:

```
% mysql -p
Enter password:        ← enter your password here
```

Command-line parameters take precedence over parameters found in your option file, so to override an option file parameter, just specify it on the command line. For example, you can list your regular MySQL username and password in the option-file for general-purpose use. Then, if you must connect on occasion as the MySQL `root` user, specify the user and password options on the command line to override the option-file values:

```
% mysql -u root -p
Enter password:        ← enter MySQL root account password here
```

To explicitly specify "no password" when there is a nonempty password in the option file, use `--skip-password` on the command line:

```
% mysql --skip-password
```

From this point on, I'll usually show commands for MySQL programs with no connection-parameter options. I assume that you'll supply any parameters that you need, either on the command line or in an option file.

Protecting option files from other users

On a multiple-user operating system such as Unix, protect the option file located in your home directory to prevent other users from reading it and finding out how to connect to MySQL using your account. Use *chmod* to make the file private by setting its mode to enable access only by yourself. Either of the following commands do this:

```
% chmod 600 .my.cnf
% chmod go-rwx .my.cnf
```

On Windows, you can use Windows Explorer to set file permissions.

1.5. Executing SQL Statements Interactively

Problem

You've started *mysql*. Now you want to send SQL statements to the MySQL server to be executed.

Solution

Just type them in, letting *mysql* know where each one ends. Or specify "one-liners" directly on the command line.

Discussion

When you invoke *mysql*, it displays a `mysql>` prompt to tell you that it's ready for input. To execute an SQL statement at the `mysql>` prompt, type it in, add a semicolon (`;`) at the end to signify the end of the statement, and press Enter. An explicit statement terminator is necessary; *mysql* doesn't interpret Enter as a terminator because you can enter a statement using multiple input lines. The semicolon is the most common terminator, but you can also use `\g` ("go") as a synonym for the semicolon. Thus, the following examples are equivalent ways of issuing the same statement, even though they are entered differently and terminated differently:

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2014-04-06 17:43:52 |
+---------------------+
mysql> SELECT
    -> NOW()\g
+---------------------+
| NOW()               |
+---------------------+
| 2014-04-06 17:43:57 |
+---------------------+
```

For the second statement, *mysql* changes the prompt from `mysql>` to `->` to let you know that it's still waiting to see the statement terminator.

The `;` and `\g` statement terminators are not part of the statement itself. They're conventions used by the *mysql* program, which recognizes these terminators and strips them from the input before sending the statement to the MySQL server.

Some statements generate output lines that are so long they take up more than one line on your terminal, which can make query results difficult to read. To avoid this problem, generate "vertical" output by terminating the statement with `\G` rather than with `;` or `\g`. The output shows column values on separate lines:

```
mysql> SHOW FULL COLUMNS FROM limbs LIKE 'thing'\G
*************************** 1. row ***************************
     Field: thing
      Type: varchar(20)
 Collation: latin1_swedish_ci
      Null: YES
       Key:
   Default: NULL
     Extra:
Privileges: select,insert,update,references
   Comment:
```

To produce vertical output for all statements executed within a session, invoke *mysql* with the `-E` (or `--vertical`) option. To produce vertical output only for those results that exceed your terminal width, use `--auto-vertical-output`.

To execute a statement directly from the command line, specify it using the `-e` (or `--execute`) option. This is useful for "one-liners." For example, to count the rows in the `limbs` table, use this command:

```
% mysql -e "SELECT COUNT(*) FROM limbs" cookbook
+----------+
| COUNT(*) |
+----------+
|       11 |
+----------+
```

To execute multiple statements, separate them with semicolons:

```
% mysql -e "SELECT COUNT(*) FROM limbs;SELECT NOW()" cookbook
+----------+
| COUNT(*) |
+----------+
|       11 |
+----------+
+---------------------+
| NOW()               |
+---------------------+
| 2014-04-06 17:43:57 |
+---------------------+
```

mysql can also read statements from a file or from another program (see Recipe 1.6).

1.6. Executing SQL Statements Read from a File or Program

Problem

You want *mysql* to read statements stored in a file so that you need not enter them manually. Or you want *mysql* to read the output from another program.

Solution

To read a file, redirect *mysql*'s input, or use the `source` command. To read from a program, use a pipe.

Discussion

By default, the *mysql* program reads input interactively from the terminal, but you can feed it statements using other input sources such as a file or program.

To create an SQL script for *mysql* to execute in batch mode, put your statements in a text file. Then invoke *mysql* and redirect its input to read from that file:

```
% mysql cookbook < file_name
```

Statements read from an input file substitute for what you'd normally enter interactively by hand, so they must be terminated with `;`, `\g`, or `\G`, just as if you were entering them manually. Interactive and batch modes do differ in default output format. For interactive mode, the default is tabular (boxed) format. For batch mode, the default is tab-delimited format. To override the default, use the appropriate command option (see Recipe 1.7).

Batch mode is convenient for executing a set of statements on repeated occasions without entering them manually each time. Batch mode makes it easy to set up *cron* jobs that run with no user intervention. SQL scripts also are useful for distributing statements to other people. That is, in fact, how I distribute SQL examples for this book. Many of the examples shown here can be run using script files available in the accompanying `recipes` distribution (see the Preface). Feed these files to *mysql* in batch mode to avoid typing statements yourself. For example, when a recipe shows a `CREATE TABLE` statement that defines a table, you'll usually find an SQL batch file in the `recipes` distribution that you can use to create (and perhaps load data into) the table. Recall that Recipe 1.2 shows the statements for creating and populating the `limbs` table. Those statements were shown as you would enter them manually, but the *tables* directory of the `recipes` distribution includes a *limbs.sql* file that contains statements to do the same thing. The file looks like this:

```
DROP TABLE IF EXISTS limbs;
CREATE TABLE limbs
(
  thing VARCHAR(20),  # what the thing is
  legs  INT,          # number of legs it has
  arms  INT           # number of arms it has
);

INSERT INTO limbs (thing,legs,arms) VALUES('human',2,2);
INSERT INTO limbs (thing,legs,arms) VALUES('insect',6,0);
INSERT INTO limbs (thing,legs,arms) VALUES('squid',0,10);
INSERT INTO limbs (thing,legs,arms) VALUES('fish',0,0);
INSERT INTO limbs (thing,legs,arms) VALUES('centipede',100,0);
INSERT INTO limbs (thing,legs,arms) VALUES('table',4,0);
INSERT INTO limbs (thing,legs,arms) VALUES('armchair',4,2);
INSERT INTO limbs (thing,legs,arms) VALUES('phonograph',0,1);
INSERT INTO limbs (thing,legs,arms) VALUES('tripod',3,0);
INSERT INTO limbs (thing,legs,arms) VALUES('Peg Leg Pete',1,2);
INSERT INTO limbs (thing,legs,arms) VALUES('space alien',NULL,NULL);
```

To execute the statements in this SQL script file, change location into the *tables* directory of the `recipes` distribution and run this command:

```
% mysql cookbook < limbs.sql
```

You'll note that the script contains a statement to drop the table if it exists before creating the table anew and loading it with data. That enables you to experiment with the table, perhaps making changes to it, confident that you can easily restore it to its baseline state any time by running the script again.

The command just shown illustrates how to specify an input file for *mysql* on the command line. Alternatively, to read a file of SQL statements from within a *mysql* session, use a `source` *`filename`* command (or `\.` *`filename`*, which is synonymous):

```
mysql> source limbs.sql;
mysql> \. limbs.sql;
```

SQL scripts can themselves include `source` or `\.` commands to include other scripts. This gives you additional flexibility, but take care to avoid source loops.

A file to be read by *mysql* need not be written by hand; it could be program generated. For example, the *mysqldump* utility generates database backups by writing a set of SQL statements that re-create the database. To reload *mysqldump* output, feed it to *mysql*. For example, you can copy a database over the network to another MySQL server like this:

```
% mysqldump cookbook > dump.sql
% mysql -h other-host.example.com cookbook < dump.sql
```

mysql can also read a pipe, so it can take output from other programs as its input. *Any* command that produces output consisting of properly terminated SQL statements can be used as an input source for *mysql*. The dump-and-reload example can be rewritten

to connect the two programs directly with a pipe, avoiding the need for an intermediary file:

```
% mysqldump cookbook | mysql -h other-host.example.com cookbook
```

Program-generated SQL also can be useful for populating a table with test data without writing the `INSERT` statements by hand. Create a program that generates the statements, then send its output to *mysql* using a pipe:

```
% generate-test-data | mysql cookbook
```

Recipe 4.6 discusses *mysqldump* further.

1.7. Controlling mysql Output Destination and Format

Problem

You want *mysql* output to go somewhere other than your screen. And you don't necessarily want the default output format.

Solution

Redirect the output to a file, or use a pipe to send the output to a program. You can also control other aspects of *mysql* output to produce tabular, tab-delimited, HTML, or XML output; suppress column headers; or make *mysql* more or less verbose.

Discussion

Unless you send *mysql* output elsewhere, it goes to your screen. To save output from *mysql* in a file, use your shell's redirection capability:

```
% mysql cookbook > outputfile
```

If you run *mysql* interactively with the output redirected, you can't see what you type, so in this case you usually also read the input from a file (or another program):

```
% mysql cookbook < inputfile > outputfile
```

To send the output to another program (for example, to mail query results to someone), use a pipe:

```
% mysql cookbook < inputfile | mail paul
```

The rest of this section shows how to control *mysql* output format.

Producing tabular or tab-delimited output

mysql chooses its default output format by whether it runs interactively or noninteractively. For interactive use, *mysql* writes output to the terminal using tabular (boxed) format:

```
% mysql
mysql> SELECT * FROM limbs WHERE legs=0;
+------------+------+------+
| thing      | legs | arms |
+------------+------+------+
| squid      |    0 |   10 |
| fish       |    0 |    0 |
| phonograph |    0 |    1 |
+------------+------+------+
3 rows in set (0.00 sec)
```

For noninteractive use (when the input or output is redirected), *mysql* writes tab-delimited output:

```
% echo "SELECT * FROM limbs WHERE legs=0" | mysql cookbook
thing   legs    arms
squid   0       10
fish    0       0
phonograph      0       1
```

To override the default output format, use the appropriate command option. Consider this command shown earlier:

```
% mysql cookbook < inputfile | mail paul
```

Because *mysql* runs noninteractively in that context, it produces tab-delimited output, which the mail recipient may find more difficult to read than tabular output. Use the `-t` (or `--table`) option to produce more readable tabular output:

```
% mysql -t cookbook < inputfile | mail paul
```

The inverse operation is to produce batch (tab-delimited) output in interactive mode. To do this, use `-B` or `--batch`.

Producing HTML or XML output

mysql generates an HTML table from each query result set if you use the `-H` (or `--html`) option. This enables you to easily produce output for inclusion in a web page that shows a query result. Here's an example (with line breaks added to make the output easier to read):

```
% mysql -H -e "SELECT * FROM limbs WHERE legs=0" cookbook
<TABLE BORDER=1>
<TR><TH>thing</TH><TH>legs</TH><TH>arms</TH></TR>
<TR><TD>squid</TD><TD>0</TD><TD>10</TD></TR>
<TR><TD>fish</TD><TD>0</TD><TD>0</TD></TR>
<TR><TD>phonograph</TD><TD>0</TD><TD>1</TD></TR>
</TABLE>
```

The first row of the table contains column headings. If you don't want a header row, see the next section for instructions.

You can save the output in a file, then view it with a web browser. For example, on Mac OS X, do this:

```
% mysql -H -e "SELECT * FROM limbs WHERE legs=0" cookbook > limbs.html
% open -a safari limbs.html
```

To generate an XML document instead of HTML, use the `-X` (or `--xml`) option:

```
% mysql -X -e "SELECT * FROM limbs WHERE legs=0" cookbook
<?xml version="1.0"?>

<resultset statement="select * from limbs where legs=0
">
  <row>
    <field name="thing">squid</field>
    <field name="legs">0</field>
    <field name="arms">10</field>
  </row>

  <row>
    <field name="thing">fish</field>
    <field name="legs">0</field>
    <field name="arms">0</field>
  </row>

  <row>
    <field name="thing">phonograph</field>
    <field name="legs">0</field>
    <field name="arms">1</field>
  </row>
</resultset>
```

You can reformat XML to suit a variety of purposes by running it through XSLT transforms. This enables you to use the same input to produce many output formats. Here is a basic transform that produces plain-text output showing the original query, plus the row values separated by commas:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- mysql-xml.xsl: interpret XML-format output from mysql client -->

<xsl:output method="text"/>

<!-- Process rows in each resultset  -->
<xsl:template match="resultset">
  <xsl:text>Query: </xsl:text>
  <xsl:value-of select="@statement"/>
  <xsl:value-of select="'&#10;'"/>
  <xsl:text>Result set:&#10;</xsl:text>
  <xsl:apply-templates select="row"/>
</xsl:template>
```

```
<!-- Process fields in each row  -->
<xsl:template match="row">
  <xsl:apply-templates select="field"/>
</xsl:template>

<!-- Display text content of each field -->
<xsl:template match="field">
  <xsl:value-of select="."/>
  <xsl:choose>
    <xsl:when test="position() != last()">
      <xsl:text>, </xsl:text> <!-- comma after all but last field -->
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="'&#10;'"/> <!-- newline after last field -->
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Use the transform like this:

```
% mysql -X -e "SELECT * FROM limbs WHERE legs=0" cookbook \
    | xsltproc mysql-xml.xsl -
Query: SELECT * FROM limbs WHERE legs=0
Result set:
squid, 0, 10
fish, 0, 0
phonograph, 0, 1
```

The `-H`, `--html` `-X`, and `--xml` options produce output only for statements that generate a result set, not for statements such as `INSERT` or `UPDATE`.

To write your own programs that generate XML from query results, see Recipe 11.9. To write web scripts that generate HTML from query results, see Chapter 18.

Suppressing column headings in query output

Tab-delimited format is convenient for generating datafiles for import into other programs. However, the first row of output for each query lists the column headings by default, which may not always be what you want. Suppose that a program named *summarize* produces descriptive statistics for a column of numbers. If you produce output from *mysql* to be used with this program, a column header row would throw off the results because *summarize* would treat it as data. To create output that contains only data values, suppress the header row with the `--skip-column-names` option:

```
% mysql --skip-column-names -e "SELECT arms FROM limbs" cookbook | summarize
```

Specifying the "silent" option (`-s` or `--silent`) twice achieves the same effect:

```
% mysql -ss -e "SELECT arms FROM limbs" cookbook | summarize
```

Specifying the output column delimiter

In noninteractive mode, *mysql* separates output columns by tabs and there is no option for specifying the output delimiter. To produce output that uses a different delimiter, postprocess *mysql* output. Suppose that you want to create an output file for use by a program that expects values to be separated by colon characters (`:`) rather than tabs. Under Unix, you can convert tabs to arbitrary delimiters by using a utility such as *tr* or *sed*. Any of the following commands change tabs to colons (*TAB* indicates where you type a tab character):

```
% mysql cookbook < inputfile  | sed -e "s/TAB/:/g" > outputfile
% mysql cookbook < inputfile  | tr "TAB" ":" > outputfile
% mysql cookbook < inputfile  | tr "\011" ":" > outputfile
```

The syntax differs among versions of *tr*; consult your local documentation. Also, some shells use the tab character for special purposes such as filename completion. For such shells, type a literal tab into the command by preceding it with Ctrl-V.

sed is more powerful than *tr* because it understands regular expressions and permits multiple substitutions. This is useful for producing output in something like comma-separated values (CSV) format, which requires three substitutions:

1. Escape any quote characters that appear in the data by doubling them, so that when you use the resulting CSV file, they won't be interpreted as column delimiters.
2. Change the tabs to commas.
3. Surround column values with quotes.

sed permits all three substitutions to be performed in a single command line:

```
% mysql cookbook < inputfile \
    | sed -e 's/"/""/g' -e 's/TAB/","/g' -e 's/^/"/' -e 's/$/"/' > outputfile
```

That's cryptic, to say the least. You can achieve the same result with other languages that may be easier to read. Here's a short Perl script that does the same thing as the *sed* command (it converts tab-delimited input to CSV output), and includes comments to document how it works:

```
#!/usr/bin/perl
# csv.pl: convert tab-delimited input to comma-separated values output
while (<>)          # read next input line
{
  s/"/""/g;         # double quotes within column values
  s/\t/","/g;       # put "," between column values
  s/^/"/;           # add " before the first value
  s/$/"/;           # add " after the last value
  print;            # print the result
}
```

If you name the script *csv.pl*, use it like this:

```
% mysql cookbook < inputfile  | perl csv.pl > outputfile
```

tr and *sed* normally are unavailable under Windows. Perl may be more suitable as a cross-platform solution because it runs under both Unix and Windows. (On Unix systems, Perl is usually preinstalled. On Windows, it is freely available for you to install.)

Another way to produce CSV output is to use the Perl Text::CSV_XS module, which was designed for that purpose. Recipe 11.5 discusses this module and uses it to construct a general-purpose file reformatter.

Controlling mysql's verbosity level

When you run *mysql* noninteractively, not only does the default output format change, but it becomes more terse. For example, *mysql* doesn't print row counts or indicate how long statements took to execute. To tell *mysql* to be more verbose, use `-v` or `--verbose`, specifying the option multiple times for increasing verbosity. Try the following commands to see how the output differs:

```
% echo "SELECT NOW()" | mysql
% echo "SELECT NOW()" | mysql -v
% echo "SELECT NOW()" | mysql -vv
% echo "SELECT NOW()" | mysql -vvv
```

The counterparts of `-v` and `--verbose` are `-s` and `--silent`, which also can be used multiple times for increased effect.

1.8. Using User-Defined Variables in SQL Statements

Problem

You want to use a value in one statement that is produced by an earlier statement.

Solution

Save the value in a user-defined variable to store it for later use.

Discussion

To save a value returned by a `SELECT` statement, assign it to a user-defined variable. This enables you to refer to it in other statements later in the same session (but not *across* sessions). User variables are a MySQL-specific extension to standard SQL. They will not work with other database engines.

To assign a value to a user variable within a `SELECT` statement, use `@var_name := value` syntax. The variable can be used in subsequent statements wherever an expression is permitted, such as in a `WHERE` clause or in an `INSERT` statement.

Here is an example that assigns a value to a user variable, then refers to that variable later. This is a simple way to determine a value that characterizes some row in a table, then select that particular row:

```
mysql> SELECT @max_limbs := MAX(arms+legs) FROM limbs;
+-----------------------------+
| @max_limbs := MAX(arms+legs) |
+-----------------------------+
|                         100 |
+-----------------------------+
mysql> SELECT * FROM limbs WHERE arms+legs = @max_limbs;
+-----------+------+------+
| thing     | legs | arms |
+-----------+------+------+
| centipede |  100 |    0 |
+-----------+------+------+
```

Another use for a variable is to save the result from `LAST_INSERT_ID()` after creating a new row in a table that has an `AUTO_INCREMENT` column:

```
mysql> SELECT @last_id := LAST_INSERT_ID();
```

`LAST_INSERT_ID()` returns the most recent `AUTO_INCREMENT` value. By saving it in a variable, you can refer to the value several times in subsequent statements, even if you issue other statements that create their own `AUTO_INCREMENT` values and thus change the value returned by `LAST_INSERT_ID()`. Recipe 13.10 discusses this technique further.

User variables hold single values. If a statement returns multiple rows, the value from the last row is assigned:

```
mysql> SELECT @name := thing FROM limbs WHERE legs = 0;
+----------------+
| @name := thing |
+----------------+
| squid          |
| fish           |
| phonograph     |
+----------------+
mysql> SELECT @name;
+------------+
| @name      |
+------------+
| phonograph |
+------------+
```

If the statement returns no rows, no assignment takes place, and the variable retains its previous value. If the variable has not been used previously, its value is `NULL`:

```
mysql> SELECT @name2 := thing FROM limbs WHERE legs < 0;
Empty set (0.00 sec)
mysql> SELECT @name2;
+--------+
| @name2 |
```

```
+--------+
| NULL   |
+--------+
```

To set a variable explicitly to a particular value, use a `SET` statement. `SET` syntax can use either `:=` or `=` as the assignment operator:

```
mysql> SET @sum = 4 + 7;
mysql> SELECT @sum;
+------+
| @sum |
+------+
|   11 |
+------+
```

You can assign a `SELECT` result to a variable, provided that you write it as a scalar subquery (a query within parentheses that returns a single value):

```
mysql> SET @max_limbs = (SELECT MAX(arms+legs) FROM limbs);
```

User variable names are not case sensitive:

```
mysql> SET @x = 1, @X = 2; SELECT @x, @X;
+------+------+
| @x   | @X   |
+------+------+
| 2    | 2    |
+------+------+
```

User variables can appear only where expressions are permitted, not where constants or literal identifiers must be provided. It's tempting to attempt to use variables for such things as table names, but it doesn't work. For example, if you try to generate a temporary table name using a variable as follows, it fails:

```
mysql> SET @tbl_name = CONCAT('tmp_tbl_', CONNECTION_ID());
mysql> CREATE TABLE @tbl_name (int_col INT);
ERROR 1064: You have an error in your SQL syntax near '@tbl_name
(int_col INT)'
```

However, you *can* generate a prepared SQL statement that incorporates `@tbl_name`, then execute the result. Recipe 4.4 shows how.

`SET` is also used to assign values to stored program parameters and local variables, and to system variables. For examples, see Chapter 9 and Recipe 22.1.

CHAPTER 2

Writing MySQL-Based Programs

2.0. Introduction

This chapter discusses how to use MySQL from within the context of a general-purpose programming language. It covers basic application programming interface (API) operations that are fundamental to and form the basis for the programming recipes developed in later chapters. These operations include connecting to the MySQL server, executing statements, and retrieving the results.

MySQL-based client programs can be written using many languages. This book covers the languages and interfaces shown in the following table (for information on obtaining the interface software, see the Preface):

Language	Interface
Perl	Perl DBI
Ruby	Ruby DBI
PHP	PDO
Python	DB API
Java	JDBC

MySQL client APIs provide these capabilities, each covered in a section of this chapter:

Connecting to the MySQL server, selecting a database, and disconnecting from the server

Every program that uses MySQL must first establish a connection to the server. Most programs also select a default database, and well-behaved MySQL programs close the connection to the server when they're done with it.

Checking for errors
: Any database operation can fail. If you should know how to find out when that occurs and why, you can take appropriate action such as terminating the program or informing the user of the problem.

Executing SQL statements and retrieving results
: The point of connecting to a database server is to execute SQL statements. Each API provides at least one way to do this, as well as methods for processing statement results.

Handling special characters and `NULL` *values in statements*
: Data values can be embedded directly in statement strings. However, some characters such as quotes and backslashes have special meaning, and their use requires certain precautions. The same is true for `NULL` values. If you handle these improperly, your programs will generate SQL statements that are erroneous or yield unexpected results. If you incorporate data from external sources into queries, your program might become subject to SQL injection attacks. Most APIs enable you to avoid these problems by using placeholders: refer to data values symbolically in a statement to be executed and supply those values separately. The API inserts them into the statement string after properly encoding any special characters or `NULL` values. Placeholders are also known as parameter markers.

Identifying `NULL` *values in result sets*
: `NULL` values are special not only when you construct statements, but also in results returned from them. Each API provides a convention for recognizing and dealing with them.

No matter which programming language you use, it's necessary to know how to perform each of the fundamental database API operations just described, so this chapter shows each operation in all five languages. Seeing how each API handles a given operation should help you see the correspondences between APIs more easily and better understand the recipes shown in the following chapters, even if they're written in a language you don't use much. (Later chapters usually implement recipes using only one or two languages.)

It may seem overwhelming to see each recipe in several languages if your interest is in only one particular API. If so, I advise you to read just the introductory recipe part that provides the general background, then go directly to the section for the language in which you're interested. Skip the other languages; should you develop an interest in them later, come back and read about them then.

This chapter also discusses the following topics, which are not directly part of the MySQL APIs but help you use them more easily:

Writing library files
: As you write program after program, you find that you carry out certain operations repeatedly. Library files enable encapsulating code for those operations so they can be performed easily from multiple scripts without repeating the code in each one. This reduces code duplication and makes your programs more portable. This chapter shows how to write a library file for each API that includes a routine for connecting to the server—one operation that every program that uses MySQL must perform. Later chapters develop additional library routines for other operations.

Additional techniques for obtaining connection parameters
: An early section on establishing connections to the MySQL server relies on connection parameters hardwired into the code. However, there are other (and better) ways to obtain parameters, ranging from storing them in a separate file to enabling the user to specify them at runtime.

To avoid manually typing in the example programs, get a copy of the `recipes` source distribution (see the Preface). Then, when an example says something like "create a file named *xyz* that contains the following information ...," you can use the corresponding file from the `recipes` distribution. Most scripts for this chapter are located under the *api* directory; library files are located in the *lib* directory.

The primary table used for examples in this chapter is named `profile`. It first appears in Recipe 2.4, which you should know in case you skip around in the chapter and wonder where it came from. See also the section at the very end of the chapter about resetting the `profile` table to a known state for use in other chapters.

The programs discussed here can be run from the command line. For instructions on invoking programs for each language covered here, read "Executing Programs from the Command Line" on the companion website (see the Preface).

Assumptions

To use the material in this chapter most effectively, make sure to satisfy these requirements:

- Install MySQL programming support for any languages that you plan to use (see the Preface).
- You should already have set up a MySQL user account for accessing the server and a database for executing SQL statements. As described in Recipe 1.1, the examples in this book use a MySQL account that has a username and password of `cbuser` and `cbpass`, and we'll connect to a MySQL server running on the local host to access

a database named `cookbook`. To create the account or the database, see the instructions in that recipe.

- The discussion here shows how to use each API language to perform database operations, but assumes a basic understanding of the language itself. If a recipe uses programming constructs with which you're unfamiliar, consult a general reference for the language of interest.
- Proper execution of some of the programs might require that you set certain environment variables. General syntax for doing so is covered in "Executing Programs from the Command Line" on the companion website (see the Preface). For details about environment variables that apply specifically to library file locations, see Recipe 2.3.

MySQL Client API Architecture

Each MySQL programming interface covered in this book uses a two-level architecture:

- The upper level provides database-independent methods that implement database access in a portable way that's the same whether you use MySQL, PostgreSQL, Oracle, or whatever.
- The lower level consists of a set of drivers, each of which implements the details for a single database system.

This two-level architecture enables application programs to use an abstract interface not tied to details specific to any particular database server. This enhances portability of your programs: to use a different database system, just select a different lower-level driver. However, perfect portability is elusive:

- The interface methods provided by the upper level of the architecture are consistent regardless of the driver you use, but it's still possible to write SQL statements that use constructs supported only by a particular server. For example, MySQL has `SHOW` statements that provide information about database and table structure, but using `SHOW` with a non-MySQL server likely will produce an error.
- Lower-level drivers often extend the abstract interface to make it more convenient to access database-specific features. For example, the MySQL driver for Perl DBI makes the most recent `AUTO_INCREMENT` value available as a database handle attribute accessible as `$dbh->{mysql_insertid}`. Such features make a program easier to write, but less portable. To use the program with another database system will require some rewriting.

Despite these factors that compromise portability to some extent, the general portability characteristics of the two-level architecture provide significant benefits for MySQL developers.

Another characteristic common to the APIs used in this book is that they are object oriented. Whether you write in Perl, Ruby, PHP, Python, or Java, the operation that connects to the MySQL server returns an object that enables you to process statements in an object-oriented manner. For example, when you connect to the database server, you get a database connection object with which to further interact with the server. The interfaces also provide objects for statements, result sets, metadata, and so forth.

Now let's see how to use these programming interfaces to perform the most fundamental MySQL operations: connecting to and disconnecting from the server.

2.1. Connecting, Selecting a Database, and Disconnecting

Problem

You need to establish a connection to the database server and shut down the connection when you're done.

Solution

Each API provides routines for connecting and disconnecting. The connection routines require that you provide parameters specifying the host on which the MySQL server is running and the MySQL account to use. You can also select a default database.

Discussion

This section shows how to perform some fundamental operations common to most MySQL programs:

Establishing a connection to the MySQL server
: Every program that uses MySQL does this, no matter which API you use. The details on specifying connection parameters vary between APIs, and some APIs provide more flexibility than others. However, there are many common parameters, such as the host on which the server is running, and the username and password of the MySQL account to use for accessing the server.

Selecting a database
: Most MySQL programs select a default database.

Disconnecting from the server
: Each API provides a way to close an open connection. It's best to do so as soon as you're done using the server. If your program holds the connection open longer than necessary, the server cannot free up resources allocated to servicing the con-

nection. It's also preferable to close the connection explicitly. If a program simply terminates, the MySQL server eventually notices, but an explicit close on the user end enables the server to perform an immediate orderly close on its end.

This section includes example programs that show how to use each API to connect to the server, select the `cookbook` database, and disconnect. The discussion for each API also indicates how to connect without selecting any default database. This might be the case if you plan to execute a statement that doesn't require a default database, such as `SHOW VARIABLES` or `SELECT VERSION()`. Or perhaps you're writing a program that enables the user to specify the database after the connection has been made.

The scripts shown here use `localhost` as the hostname. If they produce a connection error indicating that a socket file cannot be found, try changing `localhost` to `127.0.0.1`, the TCP/IP address of the local host. This tip applies throughout the book.

Perl

To write MySQL scripts in Perl, the DBI module must be installed, as well as the MySQL-specific driver module, DBD::mysql. To obtain these modules if they're not already installed, see the Preface.

The following Perl script, *connect.pl*, connects to the MySQL server, selects `cookbook` as the default database, and disconnects:

```
#!/usr/bin/perl
# connect.pl: connect to the MySQL server

use strict;
use warnings;
use DBI;

my $dsn = "DBI:mysql:host=localhost;database=cookbook";
my $dbh = DBI->connect ($dsn, "cbuser", "cbpass")
            or die "Cannot connect to server\n";
print "Connected\n";
$dbh->disconnect ();
print "Disconnected\n";
```

To try *connect.pl*, locate it under the *api* directory of the `recipes` distribution and run it from the command line. The program should print two lines indicating that it connected and disconnected successfully:

```
% perl connect.pl
Connected
Disconnected
```

For background on running Perl programs, read "Executing Programs from the Command Line" on the companion website (see the Preface).

The `use strict` line turns on strict variable checking and causes Perl to complain about any variables that are used without having been declared first. This precaution helps find errors that might otherwise go undetected.

The `use warnings` line turns on warning mode so that Perl produces warnings for any questionable constructs. Our example script has none, but it's a good idea to get in the habit of enabling warnings to catch problems that occur during the script development process. `use warnings` is similar to specifying the Perl `-w` command-line option, but provides more control over which warnings to display. (For more information, execute a *perldoc warnings* command.)

The `use DBI` statement tells Perl to load the DBI module. It's unnecessary to load the MySQL driver module (DBD::mysql) explicitly. DBI does that itself when the script connects to the database server.

The next two lines establish the connection to MySQL by setting up a data source name (DSN) and calling the DBI `connect()` method. The arguments to `connect()` are the DSN, the MySQL username and password, and any connection attributes you want to specify. The DSN is required. The other arguments are optional, although usually it's necessary to supply a username and password.

The DSN specifies which database driver to use and other options that indicate where to connect. For MySQL programs, the DSN has the format `DBI:mysql:`*`options`*. The second colon in the DSN is required even if you specify no following options.

Use the DSN components as follows:

- The first component is always `DBI`. It's not case sensitive.
- The second component tells DBI which database driver to use, and it *is* case sensitive. For MySQL, the name must be `mysql`.
- The third component, if present, is a semicolon-separated list of *`name=value`* pairs that specify additional connection options, in any order. For our purposes, the two most relevant options are `host` and `database`, to specify the hostname where the MySQL server is running and the default database.

Based on that information, the DSN for connecting to the `cookbook` database on the local host *localhost* looks like this:

```
DBI:mysql:host=localhost;database=cookbook
```

If you omit the `host` option, its default value is `localhost`. These two DSNs are equivalent:

```
DBI:mysql:host=localhost;database=cookbook
DBI:mysql:database=cookbook
```

To select no default database, omit the `database` option.

The second and third arguments of the `connect()` call are your MySQL username and password. Following the password, you can also provide a fourth argument to specify attributes that control DBI's behavior when errors occur. With no attributes, DBI by default prints error messages when errors occur but does not terminate your script. That's why *connect.pl* checks whether `connect()` returns `undef`, which indicates failure:

```
my $dbh = DBI->connect ($dsn, "cbuser", "cbpass")
            or die "Cannot connect to server\n";
```

Other error-handling strategies are possible. For example, to tell DBI to terminate the script if an error occurs in any DBI call, disable the `PrintError` attribute and enable `RaiseError` instead:

```
my $dbh = DBI->connect ($dsn, "cbuser", "cbpass",
                        {PrintError => 0, RaiseError => 1});
```

Then you need not check for errors yourself. The trade-off is that you also lose the ability to decide how your program recovers from errors. Recipe 2.2 discusses error handling further.

Another common attribute is `AutoCommit`, which sets the connection's auto-commit mode for transactions. MySQL enables this by default for new connections, but we'll set it from this point on to make the initial connection state explicit:

```
my $dbh = DBI->connect ($dsn, "cbuser", "cbpass",
                        {PrintError => 0, RaiseError => 1, AutoCommit => 1});
```

As shown, the fourth argument to `connect()` is a reference to a hash of attribute name/value pairs. An alternative way of writing this code follows:

```
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
my $dbh = DBI->connect ($dsn, "cbuser", "cbpass", $conn_attrs);
```

Use whichever style you prefer. Scripts in this book use the `$conn_attr` hashref to make `connect()` calls simpler to read.

Assuming that `connect()` succeeds, it returns a database handle that contains information about the state of the connection. (In DBI parlance, references to objects are called handles.) Later we'll see other handles such as statement handles, which are associated with particular statements. Perl DBI scripts in this book conventionally use `$dbh` and `$sth` to signify database and statement handles.

Additional connection parameters. To specify the path to a socket file for *localhost* connections on Unix, provide a `mysql_socket` option in the DSN:

```
my $dsn = "DBI:mysql:host=localhost;database=cookbook"
        . ";mysql_socket=/var/tmp/mysql.sock";
```

To specify the port number for non-*localhost* (TCP/IP) connections, provide a `port` option:

```
my $dsn = "DBI:mysql:host=127.0.0.1;database=cookbook;port=3307";
```

Ruby

To write MySQL scripts in Ruby, the DBI module must be installed, as well as the MySQL-specific driver module. To obtain these modules if they're not already installed, see the Preface.

The following Ruby script, *connect.rb*, connects to the MySQL server, selects `cookbook` as the default database, and disconnects:

```
#!/usr/bin/ruby -w
# connect.rb: connect to the MySQL server

require "dbi"

begin
  dsn = "DBI:Mysql:host=localhost;database=cookbook"
  dbh = DBI.connect(dsn, "cbuser", "cbpass")
  puts "Connected"
rescue
  puts "Cannot connect to server"
  exit(1)
end
dbh.disconnect
puts "Disconnected"
```

To try *connect.rb*, locate it under the *api* directory of the `recipes` distribution and run it from the command line. The program should print two lines indicating that it connected and disconnected successfully:

```
% ruby connect.rb
Connected
Disconnected
```

For background on running Ruby programs, read "Executing Programs from the Command Line" on the companion website (see the Preface).

The `-w` option turns on warning mode so that Ruby produces warnings for any questionable constructs. Our example script has no such constructs, but it's a good idea to get in the habit of using `-w` to catch problems that occur during the script development process.

The `require` statement tells Ruby to load the DBI module. It's unnecessary to load the MySQL driver module explicitly. DBI does that itself when the script connects to the database server.

To establish the connection, pass a data source name (DSN) and the MySQL username and password to the `connect()` method. The DSN is required. The other arguments are optional, although usually it's necessary to supply a username and password.

The DSN specifies which database driver to use and other options that indicate where to connect. For MySQL programs, the DSN typically has one of these formats:

```
DBI:Mysql:db_name:host_name
DBI:Mysql:name=value;name=value ...
```

As with Perl DBI, the second colon in the DSN is required even if you specify no following options.

Use the DSN components as follows:

- The first component is always `DBI` or `dbi`.
- The second component tells DBI which database driver to use. For MySQL, the name is `Mysql`.
- The third component, if present, is either a database name and hostname separated by a colon, or a semicolon-separated list of *`name=value`* pairs that specify additional connection options, in any order. For our purposes, the two most relevant options are `host` and `database`, to specify the hostname where the MySQL server is running and the default database.

Based on that information, the DSN for connecting to the `cookbook` database on the local host *localhost* looks like this:

```
DBI:Mysql:host=localhost;database=cookbook
```

If you omit the `host` option, its default value is `localhost`. These two DSNs are equivalent:

```
DBI:Mysql:host=localhost;database=cookbook
DBI:Mysql:database=cookbook
```

To select no default database, omit the `database` option.

Assuming that `connect()` succeeds, it returns a database handle that contains information about the state of the connection. Ruby DBI scripts in this book conventionally use `dbh` to signify a database handle.

If the `connect()` method fails, DBI raises an exception. To handle exceptions, put the statements that might fail inside a `begin` block, and use a `rescue` clause that contains the error-handling code. Exceptions that occur at the top level of a script (that is, outside

of any `begin` block) are caught by the default exception handler, which prints a stack trace and exits. Recipe 2.2 discusses error handling further.

Additional connection parameters. To specify the path to a socket file for *localhost* connections on Unix, provide a `socket` option in the DSN:

```
dsn = "DBI:Mysql:host=localhost;database=cookbook" +
        ";socket=/var/tmp/mysql.sock"
```

To specify the port number for non-*localhost* (TCP/IP) connections, provide a `port` option:

```
dsn = "DBI:Mysql:host=127.0.0.1;database=cookbook;port=3307"
```

PHP

To write PHP scripts that use MySQL, your PHP interpreter must have MySQL support compiled in. If your scripts are unable to connect to your MySQL server, check the instructions included with your PHP distribution to see how to enable MySQL support.

PHP actually has multiple extensions that enable the use of MySQL, such as `mysql`, the original (and now deprecated) MySQL extension; `mysqli`, the "MySQL improved" extension; and, more recently, the MySQL driver for the PDO (PHP Data Objects) interface. PHP scripts in this book use PDO. To obtain PHP and PDO if they're not already installed, see the Preface.

PHP scripts usually are written for use with a web server. I assume that if you use PHP that way, you can copy PHP scripts into your server's document tree, request them from your browser, and they will execute. For example, if you run Apache as the web server on the host *localhost* and you install a PHP script named *myscript.php* at the top level of the Apache document tree, you should be able to access the script by requesting this URL:

```
http://localhost/myscript.php
```

This book uses the *.php* extension (suffix) for PHP script filenames, so your web server must be configured to recognize the *.php* extension (for Apache, see Recipe 18.2). Otherwise, when you request a PHP script from your browser, the server simply sends the literal text of the script and that's what appears in your browser window. You don't want this to happen, particularly if the script contains the username and password for connecting to MySQL.

PHP scripts often are written as a mixture of HTML and PHP code, with the PHP code embedded between the special `<?php` and `?>` tags. Here is an example:

```
<html>
<head><title>A simple page</title></head>
<body>
<p>
```

```
<?php
  print ("I am PHP code, hear me roar!");
?>
</p>
</body>
</html>
```

For brevity in examples consisting entirely of PHP code, typically I'll omit the enclosing `<?php` and `?>` tags. If you see no tags in a PHP example, assume that `<?php` and `?>` surround the entire block of code that is shown. Examples that switch between HTML and PHP code do include the tags, to make it clear what is PHP code and what is not.

PHP can be configured to recognize "short" tags as well, written as `<?` and `?>`. This book does not assume that you have short tags enabled and does not use them.

The following PHP script, *connect.php*, connects to the MySQL server, selects `cookbook` as the default database, and disconnects:

```
<?php
# connect.php: connect to the MySQL server

try
{
  $dsn = "mysql:host=localhost;dbname=cookbook";
  $dbh = new PDO ($dsn, "cbuser", "cbpass");
  print ("Connected\n");
}
catch (PDOException $e)
{
  die ("Cannot connect to server\n");
}
$dbh = NULL;
print ("Disconnected\n");
?>
```

To try *connect.php*, locate it under the *api* directory of the `recipes` distribution, copy it to your web server's document tree, and request it using your browser. Alternatively, if you have a standalone version of the PHP interpreter for use from the command line, execute the script directly:

```
% php connect.php
Connected
Disconnected
```

For background on running PHP programs, read "Executing Programs from the Command Line" on the companion website (see the Preface).

`$dsn` is the data source name (DSN) that indicates how to connect to the database server. It has this general syntax:

```
driver:name=value;name=value ...
```

The *driver* value is the PDO driver type. For MySQL, this is `mysql`.

Following the driver name, semicolon-separated *name=value* pairs specify connection parameters, in any order. For our purposes, the two most relevant options are `host` and `dbname`, to specify the hostname where the MySQL server is running and the default database. To select no default database, omit the `dbname` option.

To establish the connection, invoke the `new PDO()` class constructor, passing to it the appropriate arguments. The DSN is required. The other arguments are optional, although usually it's necessary to supply a username and password. If the connection attempt succeeds, `new PDO()` returns a database-handle object that is used to access other MySQL-related methods. PHP scripts in this book conventionally use `$dbh` to signify a database handle.

If the connection attempt fails, PDO raises an exception. To handle this, put the connection attempt within a `try` block and use a `catch` block that contains the error-handling code, or just let the exception terminate your script. Recipe 2.2 discusses error handling further.

To disconnect, set the database handle to `NULL`. There is no explicit disconnect call.

Additional connection parameters. To specify the path to a socket file for *localhost* connections on Unix, provide a `unix_socket` option in the DSN:

```
$dsn = "mysql:host=localhost;dbname=cookbook"
         . ";unix_socket=/var/tmp/mysql.sock";
```

To specify the port number for non-*localhost* (TCP/IP) connections, provide a `port` option:

```
$dsn = "mysql:host=127.0.0.1;database=cookbook;port=3307";
```

Python

To write MySQL programs in Python, a module must be installed that provides MySQL connectivity for the Python DB API, also known as Python Database API Specification v2.0 (PEP 249). This book uses MySQL Connector/Python. To obtain it if it's not already installed, see the Preface.

To use the DB API, import the database driver module that you want to use (which is `mysql.connector` for MySQL programs that use Connector/Python). Then create a database connection object by calling the driver's `connect()` method. This object provides access to other DB API methods, such as the `close()` method that severs the connection to the database server.

The following Python script, *connect.py*, connects to the MySQL server, selects `cookbook` as the default database, and disconnects:

```
#!/usr/bin/python
# connect.py: connect to the MySQL server

import mysql.connector

try:
  conn = mysql.connector.connect(database="cookbook",
                                 host="localhost",
                                 user="cbuser",
                                 password="cbpass")
  print("Connected")
except:
  print("Cannot connect to server")
else:
  conn.close()
  print("Disconnected")
```

To try *connect.py*, locate it under the *api* directory of the `recipes` distribution and run it from the command line. The program should print two lines indicating that it connected and disconnected successfully:

```
% python connect.py
Connected
Disconnected
```

For background on running Python programs, read "Executing Programs from the Command Line" on the companion website (see the Preface).

The `import` line tells Python to load the `mysql.connector` module. Then the script attempts to establish a connection to the MySQL server by calling `connect()` to obtain a connection object. Python scripts in this book conventionally use `conn` to signify connection objects.

If the `connect()` method fails, Connector/Python raises an exception. To handle exceptions, put the statements that might fail inside a `try` statement and use an `except` clause that contains the error-handling code. Exceptions that occur at the top level of a script (that is, outside of any `try` statement) are caught by the default exception handler, which prints a stack trace and exits. Recipe 2.2 discusses error handling further.

The `else` clause contains statements that execute if the `try` clause produces no exception. It's used here to close the successfully opened connection.

Because the `connect()` call uses named arguments, their order does not matter. If you omit the `host` argument from the `connect()` call, its default value is `127.0.0.1`. To select no default database, omit the `database` argument or pass a `database` value of `""` (the empty string) or `None`.

Another way to connect is to specify the parameters using a Python dictionary and pass the dictionary to `connect()`:

```
conn_params = {
  "database": "cookbook",
  "host": "localhost",
  "user": "cbuser",
  "password": "cbpass",
}
conn = mysql.connector.connect(**conn_params)
print("Connected")
```

This book generally uses that style from now on.

Additional connection parameters. To specify the path to a socket file for local host connections on Unix, omit the `host` parameter and provide a `unix_socket` parameter:

```
conn_params = {
  "database": "cookbook",
  "unix_socket": "/var/tmp/mysql.sock",
  "user": "cbuser",
  "password": "cbpass",
}
conn = mysql.connector.connect(**conn_params)
print("Connected")
```

To specify the port number for TCP/IP connections, include the `host` parameter and provide an integer-valued `port` parameter:

```
conn_params = {
  "database": "cookbook",
  "host": "127.0.0.1",
  "port": 3307,
  "user": "cbuser",
  "password": "cbpass",
}
conn = mysql.connector.connect(**conn_params)
```

Java

Database programs in Java use the JDBC interface, together with a driver for the particular database engine you want to access. That is, the JDBC architecture provides a generic interface used in conjunction with a database-specific driver.

Java programming requires a Java Development Kit (JDK), and you must set your `JAVA_HOME` environment variable to the location where your JDK is installed. To write MySQL-based Java programs, you'll also need a MySQL-specific JDBC driver. Programs in this book use MySQL Connector/J. To obtain it if it's not already installed, see the Preface. For information about obtaining a JDK and setting `JAVA_HOME`, read "Executing Programs from the Command Line" on the companion website (see the Preface).

The following Java program, *Connect.java*, connects to the MySQL server, selects `cookbook` as the default database, and disconnects:

```
// Connect.java: connect to the MySQL server

import java.sql.*;

public class Connect
{
  public static void main (String[] args)
  {
    Connection conn = null;
    String url = "jdbc:mysql://localhost/cookbook";
    String userName = "cbuser";
    String password = "cbpass";

    try
    {
      Class.forName ("com.mysql.jdbc.Driver").newInstance ();
      conn = DriverManager.getConnection (url, userName, password);
      System.out.println ("Connected");
    }
    catch (Exception e)
    {
      System.err.println ("Cannot connect to server");
      System.exit (1);
    }
    if (conn != null)
    {
      try
      {
        conn.close ();
        System.out.println ("Disconnected");
      }
      catch (Exception e) { /* ignore close errors */ }
    }
  }
}
```

To try *Connect.java*, locate it under the *api* directory of the `recipes` distribution, compile it, and execute it. The `class` statement indicates the program's name, which in this case is `Connect`. The name of the file containing the program must match this name and include a *.java* extension, so the filename for the program is *Connect.java*. Compile the program using *javac*:

```
% javac Connect.java
```

If you prefer a different Java compiler, substitute its name for *javac*.

The Java compiler generates compiled byte code to produce a class file named *Connect.class*. Use the *java* program to run the class file (specified without the *.class* extension). The program should print two lines indicating that it connected and disconnected successfully:

```
% java Connect
Connected
Disconnected
```

You might need to set your `CLASSPATH` environment variable before the example program will compile and run. The value of `CLASSPATH` should include at least your current directory (`.`) and the path to the Connector/J JDBC driver. For background on running Java programs or setting `CLASSPATH`, read "Executing Programs from the Command Line" on the companion website (see the Preface).

The `import java.sql.*` statement references the classes and interfaces that provide access to the data types used to manage different aspects of your interaction with the database server. These are required for all JDBC programs.

Connecting to the server is a two-step process. First, register the database driver with JDBC by calling `Class.forName()`. The `Class.forName()` method requires a driver name; for Connector/J, use `com.mysql.jdbc.Driver`. Then call `DriverManager.getConnection()` to initiate the connection and obtain a `Connection` object that maintains information about the state of the connection. Java programs in this book conventionally use `conn` to signify connection objects.

`DriverManager.getConnection()` takes three arguments: a URL that describes where to connect and the database to use, the MySQL username, and the password. The URL string has this format:

```
jdbc:driver://host_name/db_name
```

This format follows the Java convention that the URL for connecting to a network resource begins with a protocol designator. For JDBC programs, the protocol is `jdbc`, and you'll also need a subprotocol designator that specifies the driver name (`mysql`, for MySQL programs). Many parts of the connection URL are optional, but the leading protocol and subprotocol designators are not. If you omit *`host_name`*, the default host value is `localhost`. To select no default database, omit the database name. However, you should not omit any of the slashes in any case. For example, to connect to the local host without selecting a default database, the URL is:

```
jdbc:mysql:///
```

In JDBC, you don't test method calls for return values that indicate an error. Instead, provide handlers to be called when exceptions are thrown. Recipe 2.2 discusses error handling further.

Beware of Class.forName()!

The example program *Connect.java* registers the JDBC driver like this:

```
Class.forName ("com.mysql.jdbc.Driver").newInstance ();
```

You're supposed to be able to register drivers without invoking `newInstance()`, like so:

```
Class.forName ("com.mysql.jdbc.Driver");
```

However, that call doesn't work for some Java implementations, so be sure to use `newInstance()`, or you may find yourself enacting the Java motto, "write once, debug everywhere."

Some JDBC drivers (Connector/J among them) permit you to specify the username and password as parameters at the end of the URL. In this case, omit the second and third arguments of the `getConnection()` call. Using that URL style, write the code that establishes the connection in the example program like this:

```
// connect using username and password included in URL
Connection conn = null;
String url = "jdbc:mysql://localhost/cookbook?user=cbuser&password=cbpass";

try
{
  Class.forName ("com.mysql.jdbc.Driver").newInstance ();
  conn = DriverManager.getConnection (url);
  System.out.println ("Connected");
}
```

The character that separates the `user` and `password` parameters should be `&`, not `;`.

Additional connection parameters. Connector/J does not support Unix domain socket file connections, so even connections for which the hostname is *localhost* are made via TCP/IP. To specify an explicit port number, add `:`*`port_num`* to the hostname in the connection URL:

```
String url = "jdbc:mysql://127.0.0.1:3307/cookbook";
```

2.2. Checking for Errors

Problem

Something went wrong with your program, and you don't know what.

Solution

Everyone has problems getting programs to work correctly. But if you don't anticipate problems by checking for errors, the job becomes much more difficult. Add some error-checking code so your programs can help you figure out what went wrong.

Discussion

After working through Recipe 2.1, you know how to connect to the MySQL server. It's also a good idea to know how to check for errors and how to retrieve specific error information from the API, so we cover that next. You're probably anxious to do more interesting things (such as executing statements and getting back the results), but error checking is fundamentally important. Programs sometimes fail, especially during development, and if you can't determine why failures occur, you're flying blind.

The need to check for errors is not so obvious or widely appreciated as one might hope. Many messages posted on MySQL-related mailing lists are requests for help with programs that fail for reasons unknown to the people who wrote them. Surprisingly often, people have put in *no* error checking, thus giving themselves no way to know that there was a problem or to find out what it was! Plan for failure by checking for errors so that you can take appropriate action.

When an error occurs, MySQL provides three values:

- A MySQL-specific error number
- A MySQL-specific descriptive text error message
- A five-character SQLSTATE error code defined according to the ANSI and ODBC standards

The recipes in this section show how to access this information. The example programs are deliberately designed to fail, so that the error-handling code executes. That's why they attempt to connect using a username and password of `baduser` and `badpass`.

A general debugging aid not specific to any API is to use the available logs. Check the MySQL server's query log to see what statements the server is receiving. (This requires that log to be enabled; see Recipe 22.3.) The query log might show that your program is not constructing the SQL statement string you expect. Similarly, if you run a script under a web server and it fails, check the web server's error log.

Perl

The DBI module provides two attributes that control what happens when DBI method invocations fail:

- `PrintError`, if enabled, causes DBI to print an error message using `warn()`.
- `RaiseError`, if enabled, causes DBI to print an error message using `die()`. This terminates your script.

By default, `PrintError` is enabled and `RaiseError` is disabled, so a script continues executing after printing a message if an error occurs. Either or both attributes can be specified in the `connect()` call. Setting an attribute to 1 or 0 enables or disables it, respectively. To specify either or both attributes, pass them in a hash reference as the fourth argument to the `connect()` call.

The following code sets only the `AutoCommit` attribute and uses the default settings for the error-handling attributes. If the `connect()` call fails, a warning message results, but the script continues to execute:

```
my $conn_attrs = {AutoCommit => 1};
my $dbh = DBI->connect ($dsn, "baduser", "badpass", $conn_attrs);
```

Because you really can't do much if the connection attempt fails, it's often prudent to exit instead after DBI prints a message:

```
my $conn_attrs = {AutoCommit => 1};
my $dbh = DBI->connect ($dsn, "baduser", "badpass", $conn_attrs)
            or exit;
```

To print your own error messages, leave `RaiseError` disabled and disable `PrintError` as well. Then test the results of DBI method calls yourself. When a method fails, the `$DBI::err`, `$DBI::errstr`, and `$DBI::state` variables contain the MySQL error number, a descriptive error string, and the SQLSTATE value, respectively:

```
my $conn_attrs = {PrintError => 0, AutoCommit => 1};
my $dbh = DBI->connect ($dsn, "baduser", "badpass", $conn_attrs)
            or die "Connection error: "
                    . "$DBI::errstr ($DBI::err/$DBI::state)\n";
```

If no error occurs, `$DBI::err` is 0 or `undef`, `$DBI::errstr` is the empty string or `undef`, and `$DBI::state` is empty or `00000`.

When you check for errors, access these variables immediately after invoking the DBI method that sets them. If you invoke another method before using them, DBI resets their values.

If you print your own messages, the default settings (`PrintError` enabled, `RaiseError` disabled) are not so useful. DBI prints a message automatically, then your script

prints its own message. This is redundant, as well as confusing to the person using the script.

If you enable `RaiseError`, you can call DBI methods without checking for return values that indicate errors. If a method fails, DBI prints an error and terminates your script. If the method returns, you can assume it succeeded. This is the easiest approach for script writers: let DBI do all the error checking! However, if both `PrintError` and `RaiseError` are enabled, DBI may call `warn()` and `die()` in succession, resulting in error messages being printed twice. To avoid this problem, disable `PrintError` whenever you enable `RaiseError`:

```
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
my $dbh = DBI->connect ($dsn, "baduser", "badpass", $conn_attrs);
```

This book generally uses that approach. If you don't want the all-or-nothing behavior of enabling `RaiseError` for automatic error checking versus having to do all your own checking, adopt a mixed approach. Individual handles have `PrintError` and `RaiseError` attributes that can be enabled or disabled selectively. For example, you can enable `RaiseError` globally by turning it on when you call `connect()`, and then disable it selectively on a per-handle basis.

Suppose that a script reads the username and password from the command-line arguments, and then loops while the user enters statements to be executed. In this case, you'd probably want DBI to die and print the error message automatically if the connection fails (you cannot proceed to the statement-execution loop in that case). After connecting, however, you wouldn't want the script to exit just because the user enters a syntactically invalid statement. Instead, print an error message and loop to get the next statement. The following code shows how to do this. The `do()` method used in the example executes a statement and returns `undef` to indicate an error:

```
my $user_name = shift (@ARGV);
my $password = shift (@ARGV);
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
my $dbh = DBI->connect ($dsn, $user_name, $password, $conn_attrs);
$dbh->{RaiseError} = 0; # disable automatic termination on error
print "Enter statements to execute, one per line; terminate with Control-D\n";
while (<>)              # read and execute queries
{
  $dbh->do ($_) or warn "Statement failed: $DBI::errstr ($DBI::err)\n";
}
```

If `RaiseError` is enabled, you can execute code within an `eval` block to trap errors without terminating your program. If an error occurs, `eval` returns a message in the `$@` variable:

```
eval
{
  # statements that might fail go here...
};
```

```
if ($@)
{
  print "An error occurred: $@\n";
}
```

This `eval` technique is commonly used to perform transactions (see Recipe 17.4).

Using `RaiseError` in combination with `eval` differs from using `RaiseError` alone:

- Errors terminate only the `eval` block, not the entire script.
- Any error terminates the `eval` block, whereas `RaiseError` applies only to DBI-related errors.

When you use `eval` with `RaiseError` enabled, disable `PrintError`. Otherwise, in some versions of DBI, an error may simply cause `warn()` to be called without terminating the `eval` block as you expect.

In addition to using the error-handling attributes `PrintError` and `RaiseError`, lots of information about your script's execution is available using DBI's tracing mechanism. Invoke the `trace()` method with an argument indicating the trace level. Levels 1 to 9 enable tracing with increasingly more verbose output, and level 0 disables tracing:

```
DBI->trace (1); # enable tracing, minimal output
DBI->trace (3); # elevate trace level
DBI->trace (0); # disable tracing
```

Individual database and statement handles also have `trace()` methods, so you can localize tracing to a single handle if you want.

Trace output normally goes to your terminal (or, in the case of a web script, to the web server's error log). To write trace output to a specific file, provide a second argument that indicates the filename:

```
DBI->trace (1, "/tmp/trace.out");
```

If the trace file already exists, its contents are not cleared first; trace output is appended to the end. Beware of turning on a file trace while developing a script, but forgetting to disable the trace when you put the script into production. You'll eventually find to your chagrin that the trace file has become quite large. Or worse, a filesystem will fill up, and you'll have no idea why!

Ruby

Ruby signals errors by raising exceptions and Ruby programs handle errors by catching exceptions in a `rescue` clause of a `begin` block. Ruby DBI methods raise exceptions when they fail and provide error information by means of a `DBI::DatabaseError` object. To get the MySQL error number, error message, and SQLSTATE value, access the `err`, `errstr`, and `state` methods of this object. The following example shows how to trap exceptions and access error information in a DBI script:

```
begin
  dsn = "DBI:Mysql:host=localhost;database=cookbook"
  dbh = DBI.connect(dsn, "baduser", "badpass")
  puts "Connected"
rescue DBI::DatabaseError => e
  puts "Cannot connect to server"
  puts "Error code: #{e.err}"
  puts "Error message: #{e.errstr}"
  puts "Error SQLSTATE: #{e.state}"
  exit(1)
end
```

PHP

The `new PDO()` constructor raises an exception if it fails, but other PDO methods by default indicate success or failure by their return value. To cause all PDO methods to raise exceptions for errors, use the database handle resulting from a successful connection attempt to set the error-handling mode. This enables uniform handling of all PDO errors without checking the result of every call. The following example shows how to set the error mode if the connection attempt succeeds and how to handle exceptions if it fails:

```
try
{
  $dsn = "mysql:host=localhost;dbname=cookbook";
  $dbh = new PDO ($dsn, "baduser", "badpass");
  $dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  print ("Connected\n");
}
catch (PDOException $e)
{
  print ("Cannot connect to server\n");
  print ("Error code: " . $e->getCode () . "\n");
  print ("Error message: " . $e->getMessage () . "\n");
}
```

When PDO raises an exception, the resulting `PDOException` object provides error information. The `getCode()` method returns the SQLSTATE value. The `getMessage()` method returns a string containing the SQLSTATE value, MySQL error number, and error message.

Database and statement handles also provide information when an error occurs. For either type of handle, `errorCode()` returns the SQLSTATE value and `errorInfo()` returns a three-element array containing the SQLSTATE value and a driver-specific error code and message. For MySQL, the latter two values are the error number and message string. The following example demonstrates how to get information from the exception object and the database handle:

```
try
{
```

```
  $dbh->query ("SELECT"); # malformed query
}
catch (PDOException $e)
{
  print ("Cannot execute query\n");
  print ("Error information using exception object:\n");
  print ("SQLSTATE value: " . $e->getCode () . "\n");
  print ("Error message: " . $e->getMessage () . "\n");

  print ("Error information using database handle:\n");
  print ("Error code: " . $dbh->errorCode () . "\n");
  $errorInfo = $dbh->errorInfo ();
  print ("SQLSTATE value: " . $errorInfo[0] . "\n");
  print ("Error number: " . $errorInfo[1] . "\n");
  print ("Error message: " . $errorInfo[2] . "\n");
}
```

Python

Python signals errors by raising exceptions, and Python programs handle errors by catching exceptions in the `except` clause of a `try` statement. To obtain MySQL-specific error information, name an exception class, and provide a variable to receive the information. Here's an example:

```
conn_params = {
  "database": "cookbook",
  "host": "localhost",
  "user": "baduser",
  "password": "badpass"
}

try:
  conn = mysql.connector.connect(**conn_params)
  print("Connected")
except mysql.connector.Error as e:
  print("Cannot connect to server")
  print("Error code: %s" % e.errno)
  print("Error message: %s" % e.msg)
  print("Error SQLSTATE: %s" % e.sqlstate)
```

If an exception occurs, the `errno`, `msg`, and `sqlstate` members of the exception object contain the error number, error message, and SQLSTATE values, respectively. Note that access to the `Error` class is through the driver module name.

Java

Java programs handle errors by catching exceptions. To do the minimum amount of work, print a stack trace to inform the user where the problem lies:

```
try
{
  /* ... some database operation ... */
```

```
}
catch (Exception e)
{
  e.printStackTrace ();
}
```

The stack trace shows the location of the problem but not necessarily what the problem was. Also, it may not be meaningful except to you, the program's developer. To be more specific, print the error message and code associated with an exception:

- All `Exception` objects support the `getMessage()` method. JDBC methods may throw exceptions using `SQLException` objects; these are like `Exception` objects but also support `getErrorCode()` and `getSQLState()` methods. `getErrorCode()` and `getMessage()` return the MySQL-specific error number and message string, and `getSQLState()` returns a string containing the SQLSTATE value.
- Some methods generate `SQLWarning` objects to provide information about nonfatal warnings. `SQLWarning` is a subclass of `SQLException`, but warnings are accumulated in a list rather than thrown immediately. They don't interrupt your program, and you can print them at your leisure.

The following example program, *Error.java*, demonstrates how to access error messages by printing all the error information available to it. It attempts to connect to the MySQL server and prints exception information if the attempt fails. Then it executes a statement and prints exception and warning information if the statement fails:

```
// Error.java: demonstrate MySQL error handling

import java.sql.*;

public class Error
{
  public static void main (String[] args)
  {
    Connection conn = null;
    String url = "jdbc:mysql://localhost/cookbook";
    String userName = "baduser";
    String password = "badpass";

    try
    {
      Class.forName ("com.mysql.jdbc.Driver").newInstance ();
      conn = DriverManager.getConnection (url, userName, password);
      System.out.println ("Connected");
      tryQuery (conn);    // issue a query
    }
    catch (Exception e)
    {
      System.err.println ("Cannot connect to server");
      System.err.println (e);
```

```
      if (e instanceof SQLException)  // JDBC-specific exception?
      {
        // e must be cast from Exception to SQLException to
        // access the SQLException-specific methods
        printException ((SQLException) e);
      }
    }
    finally
    {
      if (conn != null)
      {
        try
        {
          conn.close ();
          System.out.println ("Disconnected");
        }
        catch (SQLException e)
        {
          printException (e);
        }
      }
    }
  }

  public static void tryQuery (Connection conn)
  {
    try
    {
      // issue a simple query
      Statement s = conn.createStatement ();
      s.execute ("USE cookbook");
      s.close ();

      // print any accumulated warnings
      SQLWarning w = conn.getWarnings ();
      while (w != null)
      {
        System.err.println ("SQLWarning: " + w.getMessage ());
        System.err.println ("SQLState: " + w.getSQLState ());
        System.err.println ("Vendor code: " + w.getErrorCode ());
        w = w.getNextWarning ();
      }
    }
    catch (SQLException e)
    {
      printException (e);
    }
  }

  public static void printException (SQLException e)
  {
    // print general message, plus any database-specific message
```

```
      System.err.println ("SQLException: " + e.getMessage ());
      System.err.println ("SQLState: " + e.getSQLState ());
      System.err.println ("Vendor code: " + e.getErrorCode ());
    }
  }
```

2.3. Writing Library Files

Problem

You notice that you're repeating code to perform common operations in multiple programs.

Solution

Write routines to perform those operations, put them in a library file, and arrange for your programs to access the library. This enables you to write the code only once. You might need to set an environment variable so that your scripts can find the library.

Discussion

This section describes how to put code for common operations in library files. Encapsulation (or modularization) isn't really a "recipe" so much as a programming technique. Its principal benefit is that you need not repeat code in each program you write. Instead, simply call a routine that's in the library. For example, by putting the code for connecting to the `cookbook` database into a library routine, you need not write out all the parameters associated with making that connection. Simply invoke the routine from your program, and you're connected.

Connection establishment isn't the only operation you can encapsulate, of course. Later sections in this book develop other utility functions to be placed in library files. All such files, including those shown in this section, are located under the *lib* directory of the `recipes` distribution. As you write your own programs, be on the lookout for operations that you perform often and that are good candidates for inclusion in a library. Use the techniques in this section to write your own library files.

Library files have other benefits besides making it easier to write programs, such as promoting portability. If you write connection parameters directly into each program that connects to the MySQL server, you must change all those programs if you move them to another machine that uses different parameters. If instead you write your programs to connect to the database by calling a library routine, it's necessary only to modify the affected library routine, not all the programs that use it.

Code encapsulation can also improve security. If you make a private library file readable only to yourself, only scripts run by you can execute routines in the file. Or suppose

that you have some scripts located in your web server's document tree. A properly configured server executes the scripts and sends their output to remote clients. But if the server becomes misconfigured somehow, the result can be that it sends your scripts to clients as plain text, thus displaying your MySQL username and password. (And you'll probably realize it too late. Oops.) If you place the code for establishing a connection to the MySQL server in a library file located outside the document tree, those parameters won't be exposed to clients.

Be aware that if you install a library file to be readable by your web server, you don't have much security if other developers use the same server. Any of those developers can write a web script to read and display your library file because, by default, the script runs with the permissions of the web server and thus will have access to the library.

The recipes that follow demonstrate how to write, for each API, a library file that contains a routine for connecting to the `cookbook` database on the MySQL server. The calling program can use the error-checking techniques discussed in Recipe 2.2 to determine whether a connection attempt fails. The connection routine for each language returns a database handle or connection object when it succeeds or raises an exception if the connection cannot be established.

Libraries are of no utility in themselves, so the following discussion illustrates each one's use by a short "test harness" program. To use any of these harness programs as the basis for creating new programs, make a copy of the file and add your own code between the connect and disconnect calls.

Library-file writing involves not only the question of what to put in the file but also subsidiary issues such as where to install the file so it is accessible by your programs, and (on multiuser systems such as Unix) how to set its access privileges so its contents aren't exposed to people who shouldn't see it.

Choosing a library-file installation location

If you install a library file in a directory that a language processor searches by default, programs written in that language need do nothing special to access the library. However, if you install a library file in a directory that the language processor does not search by default, you must tell your scripts how to find it. There are two common ways to do this:

- Most languages provide a statement that can be used within a script to add directories to the language processor search path. This requires that you modify each script that needs the library.
- You can set an environment or configuration variable that changes the language processor search path. With this approach, each user who executes scripts that

require the library must set the appropriate variable. Alternatively, if the language processor has a configuration file, you might be able to set a parameter in the file that affects scripts globally for all users.

We'll use the second approach. For our API languages, the following table shows the relevant variables. In each case, the variable value is a directory or list of directories:

Language	Variable name	Variable type
Perl	`PERL5LIB`	Environment variable
Ruby	`RUBYLIB`	Environment variable
PHP	`include_path`	Configuration variable
Python	`PYTHONPATH`	Environment variable
Java	`CLASSPATH`	Environment variable

For general information on setting environment variables, read "Executing Programs from the Command Line" on the companion website (see the Preface). You can use those instructions to set environment variables to the values in the following discussion.

Suppose that you want to install library files in a directory that language processors do not search by default. For purposes of illustration, let's use */usr/local/lib/mcb* on Unix and *C:\lib\mcb* on Windows. (To put the files somewhere else, adjust the pathnames in the variable settings accordingly. For example, you might want to use a different directory, or you might want to put libraries for each language in separate directories.)

Under Unix, if you put Perl library files in the */usr/local/lib/mcb* directory, set the `PERL5LIB` environment variable appropriately. For a shell in the Bourne shell family (*sh*, *bash*, *ksh*), set the variable like this in the appropriate startup file:

```
export PERL5LIB=/usr/local/lib/mcb
```

For the original Bourne shell, *sh*, you may need to split this into two commands:

```
PERL5LIB=/usr/local/lib/mcb
export PERL5LIB
```

For a shell in the C shell family (*csh*, *tcsh*), set `PERL5LIB` like this in your *.login* file:

```
setenv PERL5LIB /usr/local/lib/mcb
```

Under Windows, if you put Perl library files in *C:\lib\mcb*, set `PERL5LIB` as follows:

```
PERL5LIB=C:\lib\mcb
```

In each case, the variable value tells Perl to look in the specified directory for library files, in addition to any other directories it searches by default. If you set `PERL5LIB` to

name multiple directories, the separator character between directory pathnames is colon (`:`) on Unix or semicolon (`;`) on Windows.

Specify the other environment variables (`RUBYLIB`, `PYTHONPATH`, and `CLASSPATH`) using the same syntax.

Setting these environment variables as just discussed should suffice for scripts that you run from the command line. For scripts intended to be executed by a web server, you likely must configure the server as well so that it can find the library files. See Recipe 18.2.

For PHP, the search path is defined by the value of the `include_path` variable in the *php.ini* PHP initialization file. On Unix, the file's pathname is likely */usr/lib/php.ini* or */usr/local/lib/php.ini*. Under Windows, the file is likely found in the Windows directory or under the main PHP installation directory. To determine the location, run this commmand:

```
% php --ini
```

Define the value of `include_path` in *php.ini* with a line like this:

```
include_path = "value"
```

Specify *value* using the same syntax as for environment variables that name directories. That is, it's a list of directory names, with the names separated by colons on Unix or semicolons on Windows. On Unix, if you want PHP to look for include files in the current directory and in */usr/local/lib/mcb*, set `include_path` like this:

```
include_path = ".:/usr/local/lib/mcb"
```

On Windows, to search the current directory and *C:\lib\mcb*, set `include_path` like this:

```
include_path = ".;C:\lib\mcb"
```

If PHP is running as an Apache module, restart Apache to make *php.ini* changes take effect.

Setting library-file access privileges

If you use a multiple-user system such as Unix, you must make decisions about library-file ownership and access mode:

- If a library file is private and contains code to be used only by you, place the file under your own account and make it accessible only to you. Assuming that a library file named *mylib* is already owned by you, you can make it private like this:

  ```
  % chmod 600 mylib
  ```

- If the library file is to be used only by your web server, install it in a server library directory and make it owned by and accessible only to the server user ID. You may need to be `root` to do this. For example, if the web server runs as `wwwusr`, the following commands make the file private to that user:

```
# chown wwwusr mylib
# chmod 600 mylib
```

- If the library file is public, you can place it in a location that your programming language searches automatically when it looks for libraries. (Most language processors search for libraries in some default set of directories, although this set can be influenced by setting environment variables as described previously.) You may need to be `root` to install files in one of these directories. Then you can make the file world readable:

```
# chmod 444 mylib
```

Now let's construct a library for each API. Each section here demonstrates how to write the library file itself and discusses how to use the library from within programs.

Perl

In Perl, library files are called modules and typically have an extension of *.pm* ("Perl module"). It's conventional for the basename of a module file to be the same as the identifier on the `package` line in the file. The following file, *Cookbook.pm*, implements a module named `Cookbook`:

```
package Cookbook;
# Cookbook.pm: library file with utility method for connecting to MySQL
# using the Perl DBI module

use strict;
use warnings;
use DBI;

my $db_name = "cookbook";
my $host_name = "localhost";
my $user_name = "cbuser";
my $password = "cbpass";
my $port_num = undef;
my $socket_file = undef;

# Establish a connection to the cookbook database, returning a database
# handle.  Raise an exception if the connection cannot be established.

sub connect
{
my $dsn = "DBI:mysql:host=$host_name";
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
```

```
  $dsn .= ";database=$db_name" if defined ($db_name);
  $dsn .= ";mysql_socket=$socket_file" if defined ($socket_file);
  $dsn .= ";port=$port_num" if defined ($port_num);

  return DBI->connect ($dsn, $user_name, $password, $conn_attrs);
}

1;  # return true
```

The module encapsulates the code for establishing a connection to the MySQL server into a `connect()` method, and the `package` identifier establishes a `Cookbook` namespace for the module. To invoke the `connect()` method, use the module name:

```
$dbh = Cookbook::connect ();
```

The final line of the module file is a statement that trivially evaluates to true. (If the module doesn't return a true value, Perl assumes that something is wrong with it and exits.)

Perl locates library files by searching the list of directories named in its `@INC` array. To check the default value of this variable on your system, invoke Perl as follows at the command line:

```
% perl -V
```

The last part of the output from the command shows the directories listed in `@INC`. If you install a library file in one of those directories, your scripts will find it automatically. If you install the module somewhere else, tell your scripts where to find it by setting the `PERL5LIB` environment variable, as discussed in the introductory part of this recipe.

After installing the *Cookbook.pm* module, try it from a test harness script, *harness.pl*:

```
#!/usr/bin/perl
# harness.pl: test harness for Cookbook.pm library

use strict;
use warnings;
use Cookbook;

my $dbh;
eval
{
  $dbh = Cookbook::connect ();
  print "Connected\n";
};
die "$@" if $@;
$dbh->disconnect ();
print "Disconnected\n";
```

harness.pl has no `use DBI` statement. It's unnecessary because the `Cookbook` module itself imports DBI; any script that uses `Cookbook` also gains access to DBI.

If you don't catch connection errors explicitly with `eval`, you can write the script body more simply:

```
my $dbh = Cookbook::connect ();
print "Connected\n";
$dbh->disconnect ();
print "Disconnected\n";
```

In this case, Perl catches any connection exception and terminates the script after printing the error message generated by the `connect()` method.

Ruby

The following Ruby library file, *Cookbook.rb*, defines a `Cookbook` class that implements a `connect` class method:

```
# Cookbook.rb: library file with utility method for connecting to MySQL
# using the Ruby DBI module

require "dbi"

# Establish a connection to the cookbook database, returning a database
# handle.  Raise an exception if the connection cannot be established.

class Cookbook
  @@host_name = "localhost"
  @@db_name = "cookbook"
  @@user_name = "cbuser"
  @@password = "cbpass"

  # Class method for connecting to server to access the
  # cookbook database; returns a database handle object.

  def Cookbook.connect
    return DBI.connect("DBI:Mysql:host=#{@@host_name};database=#{@@db_name}",
                       @@user_name, @@password)
  end
end
```

The `connect` method is defined in the library as `Cookbook.connect` because Ruby class methods are defined as *`class_name.method_name`*.

Ruby locates library files by searching the list of directories named in its `$LOAD_PATH` variable (also known as `$:`), which is an array. To check the default value of this variable on your system, use interactive Ruby to execute this statement:

```
% irb
>> puts $LOAD_PATH
```

If you install a library file in one of those directories, your scripts will find it automatically. If you install the file somewhere else, tell your scripts where to find it by setting the `RUBYLIB` environment variable, as discussed in the introductory part of this recipe.

After installing the *Cookbook.rb* library file, try it from a test harness script, *harness.rb*:

```
#!/usr/bin/ruby -w
# harness.rb: test harness for Cookbook.rb library

require "Cookbook"

begin
  dbh = Cookbook.connect
  print "Connected\n"
rescue DBI::DatabaseError => e
  puts "Cannot connect to server"
  puts "Error code: #{e.err}"
  puts "Error message: #{e.errstr}"
  exit(1)
end
dbh.disconnect
print "Disconnected\n"
```

harness.rb has no `require` statement for the DBI module. It's unnecessary because the `Cookbook` module itself imports DBI; any script that imports `Cookbook` also gains access to DBI.

If you want a script to die if an error occurs without checking for an exception yourself, write the script body like this:

```
dbh = Cookbook.connect
print "Connected\n"
dbh.disconnect
print "Disconnected\n"
```

PHP

PHP library files are written like regular PHP scripts. A *Cookbook.php* file that implements a `Cookbook` class with a `connect()` method looks like this:

```
<?php
# Cookbook.php: library file with utility method for connecting to MySQL
# using the PDO module

class Cookbook
{
  public static $host_name = "localhost";
  public static $db_name = "cookbook";
  public static $user_name = "cbuser";
  public static $password = "cbpass";

  # Establish a connection to the cookbook database, returning a database
  # handle.  Raise an exception if the connection cannot be established.
  # In addition, cause exceptions to be raised for errors.

  public static function connect ()
  {
```

```
    $dsn = "mysql:host=" . self::$host_name . ";dbname=" . self::$db_name;
    $dbh = new PDO ($dsn, self::$user_name, self::$password);
    $dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    return ($dbh);
  }

} # end Cookbook
?>
```

The `connect()` routine within the class is declared using the `static` keyword to make it a class method rather than an instance method. This designates it as directly callable without instantiating an object through which to invoke it.

The `new PDO()` constructor raises an exception if the connection attempt fails. Following a successful attempt, `connect()` sets the error-handling mode so that other PDO calls raise exceptions for failure as well. This way, individual calls need not be tested for an error return value.

Although most PHP examples throughout this book don't show the `<?php` and `?>` tags, I've shown them as part of *Cookbook.php* here to emphasize that library files must enclose all PHP code within those tags. The PHP interpreter makes no assumptions about the contents of a library file when it begins parsing it because you might include a file that contains nothing but HTML. Therefore, you must use `<?php` and `?>` to specify explicitly which parts of the library file should be considered as PHP code rather than as HTML, just as you do in the main script.

PHP looks for libraries by searching the directories named in the `include_path` variable in the PHP initialization file, as described in the introductory part of this recipe.

PHP scripts often are placed in the document tree of your web server, and clients can request them directly. For PHP library files, I recommend that you place them somewhere outside the document tree, especially if (like *Cookbook.php*) they contain a username and password.

After installing *Cookbook.php* in one of the `include_path` directories, try it from a test harness script, *harness.php*:

```
<?php
# harness.php: test harness for Cookbook.php library

require_once "Cookbook.php";

try
{
  $dbh = Cookbook::connect ();
  print ("Connected\n");
}
```

```
catch (PDOException $e)
{
  print ("Cannot connect to server\n");
  print ("Error code: " . $e->getCode () . "\n");
  print ("Error message: " . $e->getMessage () . "\n");
  exit (1);
}
$dbh = NULL;
print ("Disconnected\n");
?>
```

The `require_once` statement accesses the *Cookbook.php* file that is required to use the `Cookbook` class. `require_once` is one of several PHP file-inclusion statements:

- `require` and `include` instruct PHP to read the named file. They are similar, but `require` terminates the script if the file cannot be found; `include` produces only a warning.
- `require_once` and `include_once` are like `require` and `include` except that if the file has already been read, its contents are not processed again. This is useful for avoiding multiple-declaration problems that can easily occur when library files include other library files.

Python

Python libraries are written as modules and referenced from scripts using `import` statements. To create a method for connecting to MySQL, write a module file, *cookbook.py* (Python module names should be lowercase):

```
# cookbook.py: library file with utility method for connecting to MySQL
# using the Connector/Python module

import mysql.connector

conn_params = {
  "database": "cookbook",
  "host": "localhost",
  "user": "cbuser",
  "password": "cbpass",
}

# Establish a connection to the cookbook database, returning a connection
# object.  Raise an exception if the connection cannot be established.

def connect():
  return mysql.connector.connect(**conn_params)
```

The filename basename determines the module name, so the module is called `cookbook`. Module methods are accessed through the module name; thus, import the `cookbook` module and invoke its `connect()` method like this:

```
import cookbook

conn = cookbook.connect();
```

The Python interpreter searches for modules in directories named in the `sys.path` variable. To check the default value of `sys.path` on your system, run Python interactively and enter a few commands:

```
% python
>>> import sys
>>> sys.path
```

If you install *cookbook.py* in one of the directories named by `sys.path`, your scripts will find it with no special handling. If you install *cookbook.py* somewhere else, you must set the `PYTHONPATH` environment variable, as discussed in the introductory part of this recipe.

After installing the *cookbook.py* library file, try it from a test harness script, *harness.py*:

```
#!/usr/bin/python
# harness.py: test harness for cookbook.py library

import mysql.connector
import cookbook

try:
  conn = cookbook.connect()
  print("Connected")
except mysql.connector.Error as e:
  print("Cannot connect to server")
  print("Error code: %s" % e.errno)
  print("Error message: %s" % e.msg)
else:
  conn.close()
  print("Disconnected")
```

The *cookbook.py* file imports the `mysql.connector` module, but a script that imports `cookbook` does not thereby gain access to `mysql.connector`. If the script needs Connector/Python-specific information (such as `mysql.connector.Error`), the script itself must import `mysql.connector`.

If you want a script to die if an error occurs without checking for an exception yourself, write the script body like this:

```
conn = cookbook.connect()
print("Connected")
conn.close()
print("Disconnected")
```

Java

Java library files are similar to Java programs in most ways:

- The `class` line in the source file indicates a class name.
- The file should have the same name as the class (with a *.java* extension).
- Compile the *.java* file to produce a *.class* file.

Java library files also differ from Java programs in some ways:

- Unlike regular program files, Java library files have no `main()` function.
- A library file should begin with a `package` identifier that specifies the position of the class within the Java namespace.

A common convention for Java package identifiers is to use the domain of the code author as a prefix; this helps make identifiers unique and avoids conflict with classes written by other authors. Domain names proceed right to left from more general to more specific within the domain namespace, whereas the Java class namespace proceeds left to right from general to specific. Thus, to use a domain as the prefix for a package name within the Java class namespace, it's necessary to reverse it. For example, my domain is *kitebird.com*, so if I write a library file and place it under `mcb` within my domain's namespace, the library begins with a `package` statement like this:

```
package com.kitebird.mcb;
```

Java packages developed for this book are placed within the `com.kitebird.mcb` namespace to ensure their uniqueness in the package namespace.

The following library file, *Cookbook.java*, defines a `Cookbook` class that implements a `connect()` method for connecting to the `cookbook` database. `connect()` returns a `Connection` object if it succeeds and throws an exception otherwise. To help the caller deal with failures, the `Cookbook` class also defines `getErrorMessage()` and `printErrorMessage()` utility methods that return the error message as a string and print it to `System.err`, respectively:

```
// Cookbook.java: library file with utility methods for connecting to MySQL
// using MySQL Connector/J and for handling exceptions

package com.kitebird.mcb;

import java.sql.*;

public class Cookbook
{
  // Establish a connection to the cookbook database, returning
  // a connection object.  Throw an exception if the connection
  // cannot be established.

  public static Connection connect () throws Exception
  {
    String url = "jdbc:mysql://localhost/cookbook";
```

```
    String user = "cbuser";
    String password = "cbpass";

    Class.forName ("com.mysql.jdbc.Driver").newInstance ();
    return (DriverManager.getConnection (url, user, password));
  }

  // Return an error message as a string

  public static String getErrorMessage (Exception e)
  {
    StringBuffer s = new StringBuffer ();
    if (e instanceof SQLException)  // JDBC-specific exception?
    {
      // print general message, plus any database-specific message
      s.append ("Error message: " + e.getMessage () + "\n");
      s.append ("Error code: " + ((SQLException) e).getErrorCode () + "\n");
    }
    else
    {
      s.append (e + "\n");
    }
    return (s.toString ());
  }

  // Get the error message and print it to System.err

  public static void printErrorMessage (Exception e)
  {
    System.err.println (Cookbook.getErrorMessage (e));
  }
}
```

The routines within the class are declared using the `static` keyword, which makes them class methods rather than instance methods. That is done here because the class is used directly rather than creating an object from it and invoking the methods through the object.

To use the *Cookbook.java* file, compile it to produce *Cookbook.class*, then install the class file in a directory that corresponds to the package identifier. This means that *Cookbook.class* should be installed in a directory named *com/kitebird/mcb* (Unix) or *com\kitebird\mcb* (Windows) that is located under some directory named in your `CLASSPATH` setting. For example, if `CLASSPATH` includes */usr/local/lib/mcb* under Unix, you can install *Cookbook.class* in the */usr/local/lib/mcb/com/kitebird/mcb* directory. (For more information about the `CLASSPATH` variable, see the Java discussion in Recipe 2.1.)

To use the `Cookbook` class from within a Java program, import it and invoke the `Cookbook.connect()` method. The following test harness program, *Harness.java*, shows how to do this:

```
// Harness.java: test harness for Cookbook library class

import java.sql.*;
import com.kitebird.mcb.Cookbook;

public class Harness
{
  public static void main (String[] args)
  {
    Connection conn = null;
    try
    {
      conn = Cookbook.connect ();
      System.out.println ("Connected");
    }
    catch (Exception e)
    {
      Cookbook.printErrorMessage (e);
      System.exit (1);
    }
    finally
    {
      if (conn != null)
      {
        try
        {
          conn.close ();
          System.out.println ("Disconnected");
        }
        catch (Exception e)
        {
          String err = Cookbook.getErrorMessage (e);
          System.out.println (err);
        }
      }
    }
  }
}
```

Harness.java also shows how to use the error message utility methods from the `Cookbook` class when a MySQL-related exception occurs:

- `printErrorMessage()` takes the exception object and uses it to print an error message to `System.err`.
- `getErrorMessage()` returns the error message as a string. You can display the message yourself, write it to a logfile, or whatever.

2.4. Executing Statements and Retrieving Results

Problem

You want a program to send an SQL statement to the MySQL server and retrieve its result.

Solution

Some statements return only a status code; others return a result set (a set of rows). Some APIs provide different methods for executing each type of statement. If so, use the appropriate method for the statement to be executed.

Discussion

You can execute two general categories of SQL statements. Some retrieve information from the database; others change that information. Statements in the two categories are handled differently. In addition, some APIs provide multiple routines for executing statements, complicating matters further. Before we get to examples demonstrating how to execute statements from within each API, I'll describe the database table the examples use, and then further discuss the two statement categories and outline a general strategy for processing statements in each category.

In Chapter 1, we created a table named `limbs` to try some sample statements. In this chapter, we'll use a different table named `profile`. It's based on the idea of a "buddy list," that is, the set of people we like to keep in touch with while we're online. The table definition looks like this:

```
CREATE TABLE profile
(
  id    INT UNSIGNED NOT NULL AUTO_INCREMENT,
  name  VARCHAR(20) NOT NULL,
  birth DATE,
  color ENUM('blue','red','green','brown','black','white'),
  foods SET('lutefisk','burrito','curry','eggroll','fadge','pizza'),
  cats  INT,
  PRIMARY KEY (id)
);
```

The `profile` table indicates the things that are important to us about each buddy: name, age, favorite color, favorite foods, and number of cats—obviously one of those goofy tables used only for examples in a book! (Actually, it's not that goofy. The table uses several different data types for its columns, and these come in handy to illustrate how to solve problems that pertain to specific data types.)

The table also includes an `id` column containing unique values so that we can distinguish one row from another, even if two buddies have the same name. `id` and `name` are declared

as NOT NULL because they're each required to have a value. The other columns are implicitly permitted to be NULL (and that is also their default value) because we might not know the value to assign them for any given individual. That is, NULL signifies "unknown."

Notice that although we want to keep track of age, there is no age column in the table. Instead, there is a birth column of DATE type. Ages change, so if we store age values, we'd have to keep updating them. Storing birth dates is better: they don't change and can be used to calculate age any time (see Recipe 6.13). color is an ENUM column; color values can be any one of the listed values. foods is a SET, which permits the value to be any combination of the individual set members. That way we can record multiple favorite foods for any buddy.

To create the table, use the *profile.sql* script in the *tables* directory of the recipes distribution. Change location into that directory, then run this command:

```
% mysql cookbook < profile.sql
```

The script also loads sample data into the table. You can experiment with the table, then restore it if you change its contents by running the script again. (See the final section of this chapter on the importance of restoring the profile table after modifying it.)

The contents of the profile table as loaded by the *profile.sql* script look like this:

```
mysql> SELECT * FROM profile;
+----+---------+------------+-------+----------------------+------+
| id | name    | birth      | color | foods                | cats |
+----+---------+------------+-------+----------------------+------+
|  1 | Sybil   | 1970-04-13 | black | lutefisk,fadge,pizza |    0 |
|  2 | Nancy   | 1969-09-30 | white | burrito,curry,eggroll |    3 |
|  3 | Ralph   | 1973-11-02 | red   | eggroll,pizza        |    4 |
|  4 | Lothair | 1963-07-04 | blue  | burrito,curry        |    5 |
|  5 | Henry   | 1965-02-14 | red   | curry,fadge          |    1 |
|  6 | Aaron   | 1968-09-17 | green | lutefisk,fadge       |    1 |
|  7 | Joanna  | 1952-08-20 | green | lutefisk,fadge       |    0 |
|  8 | Stephen | 1960-05-01 | white | burrito,pizza        |    0 |
+----+---------+------------+-------+----------------------+------+
```

Although most of the columns in the profile table permit NULL values, none of the rows in the sample dataset actually contain NULL yet. (I want to defer the complications of NULL value processing to Recipes 2.5 and 2.7.)

SQL statement categories

SQL statements can be grouped into two broad categories, depending on whether they return a result set (a set of rows):

- Statements that return no result set, such as INSERT, DELETE, or UPDATE. As a general rule, statements of this type generally change the database in some way. There are

some exceptions, such as `USE db_name`, which changes the default (current) database for your session without making any changes to the database itself. The example data-modifying statement used in this section is an `UPDATE`:

```
UPDATE profile SET cats = cats+1 WHERE name = 'Sybil'
```

We'll cover how to execute this statement and determine the number of rows that it affects.

- Statements that return a result set, such as `SELECT`, `SHOW`, `EXPLAIN`, or `DESCRIBE`. I refer to such statements generically as `SELECT` statements, but you should understand that category to include any statement that returns rows. The example row-retrieval statement used in this section is a `SELECT`:

  ```
  SELECT id, name, cats FROM profile
  ```

  We'll cover how to execute this statement, fetch the rows in the result set, and determine the number of rows and columns in the result set. (To get information such as the column names or data types, access the result set metadata. That's Recipe 10.2.)

The first step in processing an SQL statement is to send it to the MySQL server for execution. Some APIs (those for Perl, Ruby, and Java, for example) recognize a distinction between the two categories of statements and provide separate calls for executing them. Other APIs (such as the one for Python) have a single call used for all statements. However, one thing all APIs have in common is that no special character indicates the end of the statement. No terminator is necessary because the end of the statement string terminates it. This differs from executing statements in the *mysql* program, where you terminate statements using a semicolon (`;`) or `\g`. (It also differs from how this book usually includes semicolons in examples to make it clear where statements end.)

When you send a statement to the server, be prepared to handle errors if it did not execute successfully. *Do not neglect this!* If a statement fails and you proceed on the basis that it succeeded, your program won't work. For the most part, this section does not show error-checking code, but that is for brevity. The sample scripts in the `recipes` distribution from which the examples are taken do include error handling, based on the techniques illustrated in Recipe 2.2.

If a statement does execute without error, your next step depends on the statement type. If it's one that returns no result set, there's nothing else to do, unless you want to check how many rows were affected. If the statement does return a result set, fetch its rows, then close the result set. In a context where you don't know whether a statement returns a result set, Recipe 10.3 discusses how to tell.

Perl

The Perl DBI module provides two basic approaches to SQL statement execution, depending on whether you expect to get back a result set. For a statement such as `IN`

SERT or UPDATE that returns no result set, use the database handle do() method. It executes the statement and returns the number of rows affected by it, or undef if an error occurs. If Sybil gets a new cat, the following statement increments her cats count by one:

```
my $count = $dbh->do ("UPDATE profile SET cats = cats+1
                       WHERE name = 'Sybil'");
if ($count)   # print row count if no error occurred
{
  $count += 0;
  print "Number of rows updated: $count\n";
}
```

If the statement executes successfully but affects no rows, do() returns a special value, "0E0" (the value zero in scientific notation, expressed as a string). "0E0" can be used for testing the execution status of a statement because it is true in Boolean contexts (unlike undef). For successful statements, it can also be used when counting how many rows were affected because it is treated as the number zero in numeric contexts. Of course, if you print that value as is, you'll print "0E0", which might look odd to people who use your program. The preceding example makes sure that doesn't happen by adding zero to the value to coerce it to numeric form so that it displays as 0. Alternatively, use printf with a %d format specifier to cause an implicit numeric conversion:

```
if ($count)   # print row count if no error occurred
{
  printf "Number of rows updated: %d\n", $count;
}
```

If RaiseError is enabled, your script terminates automatically for DBI-related errors, so you need not check $count to find out whether do() failed and consequently can simplify the code:

```
my $count = $dbh->do ("UPDATE profile SET cats = cats+1
                       WHERE name = 'Sybil'");
printf "Number of rows updated: %d\n", $count;
```

To process a statement such as SELECT that does return a result set, use a different approach that involves these steps:

1. Specify the statement to be executed by calling prepare() using the database handle. prepare() returns a statement handle to use with all subsequent operations on the statement. (If an error occurs, the script terminates if RaiseError is enabled; otherwise, prepare() returns undef.)
2. Call execute() to execute the statement and generate the result set.
3. Loop to fetch the rows returned by the statement. DBI provides several methods for this; we cover them shortly.

4. If you don't fetch the entire result set, release resources associated with it by calling `finish()`.

The following example illustrates these steps, using `fetchrow_array()` as the row-fetching method and assuming that `RaiseError` is enabled so that errors terminate the script:

```
my $sth = $dbh->prepare ("SELECT id, name, cats FROM profile");
$sth->execute ();
my $count = 0;
while (my @val = $sth->fetchrow_array ())
{
  print "id: $val[0], name: $val[1], cats: $val[2]\n";
  ++$count;
}
$sth->finish ();
print "Number of rows returned: $count\n";
```

The row array size indicates the number of columns in the result set.

The row-fetching loop just shown is followed by a call to `finish()`, which closes the result set and tells the server to free any resources associated with it. If you fetch every row in the set, DBI notices when you reach the end and releases the resources for you. Thus, the example could omit the `finish()` call without ill effect.

As the example illustrates, to determine how many rows a result set contains, count them while fetching them. Do not use the DBI `rows()` method for this purpose. The DBI documentation discourages this practice because `rows()` is not necessarily reliable for `SELECT` statements—due not to a deficiency in DBI, but to differences in behavior among database engines.

DBI has several methods that fetch a row at a time. The one used in the preceding example, `fetchrow_array()`, returns an array containing the next row, or an empty list when there are no more rows. Array elements are present in the order named in the `SELECT` statement. Access them as `$val[0]`, `$val[1]`, and so forth.

The `fetchrow_array()` method is most useful for statements that explicitly name the columns to select. (With `SELECT *`, there are no guarantees about the positions of columns within the array.)

`fetchrow_arrayref()` is like `fetchrow_array()`, except that it returns a reference to the array, or `undef` when there are no more rows. As with `fetchrow_array()`, array elements are present in the order named in the statement. Access them as `$ref->[0]`, `$ref->[1]`, and so forth:

```
while (my $ref = $sth->fetchrow_arrayref ())
{
  print "id: $ref->[0], name: $ref->[1], cats: $ref->[2]\n";
}
```

`fetchrow_hashref()` returns a reference to a hash structure, or `undef` when there are no more rows:

```
while (my $ref = $sth->fetchrow_hashref ())
{
  print "id: $ref->{id}, name: $ref->{name}, cats: $ref->{cats}\n";
}
```

To access the elements of the hash, use the names of the columns selected by the statement (`$ref->{id}`, `$ref->{name}`, and so forth). `fetchrow_hashref()` is particularly useful for `SELECT *` statements because you can access elements of rows without knowing anything about the order in which columns are returned. You need know only their names. On the other hand, it's more expensive to set up a hash than an array, so `fetchrow_hashref()` is slower than `fetchrow_array()` or `fetchrow_arrayref()`. It's also possible to "lose" row elements if they have the same name because column names must be unique. Same-name columns are not uncommon for joins between tables. For solutions to this problem, see Recipe 14.10.

In addition to the statement execution methods just described, DBI provides several high-level retrieval methods that execute a statement and return the result set in a single operation. All are database-handle methods that create and dispose of the statement handle internally before returning the result set. The methods differ in the form in which they return the result. Some return the entire result set, others return a single row or column of the set, as summarized in the following table:

Method	Return value
`selectrow_array()`	First row of result set as an array
`selectrow_arrayref()`	First row of result set as a reference to an array
`selectrow_hashref()`	First row of result set as a reference to a hash
`selectcol_arrayref()`	First column of result set as a reference to an array
`selectall_arrayref()`	Entire result set as a reference to an array of array references
`selectall_hashref()`	Entire result set as a reference to a hash of hash references

Most of these methods return a reference. The exception is `selectrow_array()`, which selects the first row of the result set and returns an array or a scalar, depending on how you call it. In array context, `selectrow_array()` returns the entire row as an array (or the empty list if no row was selected). This is useful for statements from which you expect to obtain only a single row. The return value can be used to determine the result set size. The column count is the number of elements in the array, and the row count is 1 or 0:

```
my @val = $dbh->selectrow_array ("SELECT name, birth, foods FROM profile
                                  WHERE id = 3");
my $ncols = @val;
my $nrows = $ncols ? 1 : 0;
```

You can also invoke `selectrow_array()` in scalar context, in which case it returns only the first column from the row (especially convenient for statements that return a single value):

```
my $buddy_count = $dbh->selectrow_array ("SELECT COUNT(*) FROM profile");
```

If a statement returns no result, `selectrow_array()` returns an empty array or `undef`, depending on whether you call it in array or scalar context.

`selectrow_arrayref()` and `selectrow_hashref()` select the first row of the result set and return a reference to it, or `undef` if no row was selected. To access the column values, treat the reference the same way you treat the return value from `fetchrow_arrayref()` or `fetchrow_hashref()`. The reference also provides the row and column counts:

```
my $ref = $dbh->selectrow_arrayref ($stmt);
my $ncols = defined ($ref) ? @{$ref} : 0;
my $nrows = $ncols ? 1 : 0;

my $ref = $dbh->selectrow_hashref ($stmt);
my $ncols = defined ($ref) ? keys (%{$ref}) : 0;
my $nrows = $ncols ? 1 : 0;
```

`selectcol_arrayref()` returns a reference to a single-column array representing the first column of the result set. Assuming a non-`undef` return value, access elements of the array as `$ref->[`*i*`]` for the value from row *i*. The number of rows is the number of elements in the array, and the column count is 1 or 0:

```
my $ref = $dbh->selectcol_arrayref ($stmt);
my $nrows = defined ($ref) ? @{$ref} : 0;
my $ncols = $nrows ? 1 : 0;
```

`selectall_arrayref()` returns a reference to an array containing an element for each row of the result. Each element is a reference to an array. To access row *i* of the result set, use `$ref->[`*i*`]` to get a reference to the row. Then treat the row reference the same way as a return value from `fetchrow_arrayref()` to access individual column values in the row. The result set row and column counts are available as follows:

```
my $ref = $dbh->selectall_arrayref ($stmt);
my $nrows = defined ($ref) ? @{$ref} : 0;
my $ncols = $nrows ? @{$ref->[0]} : 0;
```

`selectall_hashref()` returns a reference to a hash, each element of which is a hash reference to a row of the result. To call it, specify an argument that indicates which column to use for hash keys. For example, if you retrieve rows from the `profile` table, the primary key is the `id` column:

```
my $ref = $dbh->selectall_hashref ("SELECT * FROM profile", "id");
```

Access rows using the keys of the hash. For a row that has a key column value of `12`, the hash reference for the row is `$ref->{12}`. That row value is keyed on column names,

which you can use to access individual column elements (for example, `$ref->{12}->{name}`). The result set row and column counts are available as follows:

```
my @keys = defined ($ref) ? keys (%{$ref}) : ();
my $nrows = scalar (@keys);
my $ncols = $nrows ? keys (%{$ref->{$keys[0]}}) : 0;
```

The `selectall_XXX()` methods are useful when you need to process a result set more than once because Perl DBI provides no way to "rewind" a result set. By assigning the entire result set to a variable, you can iterate through its elements multiple times.

Take care when using the high-level methods if you have `RaiseError` disabled. In that case, a method's return value may not enable you to distinguish an error from an empty result set. For example, if you call `selectrow_array()` in scalar context to retrieve a single value, an `undef` return value is ambiguous because it may indicate any of three things: an error, an empty result set, or a result set consisting of a single `NULL` value. To test for an error, check the value of `$DBI::errstr`, `$DBI::err`, or `$DBI::state`.

Ruby

As with Perl DBI, Ruby DBI provides two approaches to SQL statement execution. With either approach, if a statement-execution method fails with an error, it raises an exception.

For statements such as `INSERT` or `UPDATE` that return no result set, invoke the `do` database-handle method. Its return value indicates the number of rows affected:

```
count = dbh.do("UPDATE profile SET cats = cats+1 WHERE name = 'Sybil'")
puts "Number of rows updated: #{count}"
```

For statements such as `SELECT` that return a result set, invoke the `execute` database-handle method. `execute` returns a statement handle for fetching result set rows. The statement handle has several methods of its own that enable row fetching in different ways. After you are done with the statement handle, invoke its `finish` method. (Call `finish` for every statement handle that you create, unlike Perl DBI where `finish` need be invoked only if you fetch a partial result set.) To determine the number of rows in the result set, count them as you fetch them.

The following example executes a `SELECT` statement and uses the statement handle's `fetch` method in a `while` loop:

```
count = 0
sth = dbh.execute("SELECT id, name, cats FROM profile")
while row = sth.fetch do
  printf "id: %s, name: %s, cats: %s\n", row[0], row[1], row[2]
  count += 1
end
sth.finish
puts "Number of rows returned: #{count}"
```

`row.size` tells you the number of columns in the result set.

`fetch` can also be used as an iterator that returns each row in turn:

```
sth.fetch do |row|
  printf "id: %s, name: %s, cats: %s\n", row[0], row[1], row[2]
end
sth.finish
```

In iterator context (such as just shown), the `each` method is a synonym for `fetch`.

The `fetch` method returns `DBI::Row` objects. Column values within the row are accessible by position, beginning with 0, as just shown, or by name:

```
sth.fetch do |row|
  printf "id: %s, name: %s, cats: %s\n",
         row["id"], row["name"], row["cats"]
end
sth.finish
```

To fetch all rows at once, use `fetch_all`, which returns an array of `DBI::Row` objects:

```
sth = dbh.execute("SELECT id, name, cats FROM profile")
rows = sth.fetch_all
sth.finish
rows.each do |row|
  printf "id: %s, name: %s, cats: %s\n",
         row["id"], row["name"], row["cats"]
end
```

To fetch each row as a hash keyed on column names, use the `fetch_hash` method. It can be called in a loop or used as an iterator. The following example shows the iterator approach:

```
sth.fetch_hash do |row|
  printf "id: %s, name: %s, cats: %s\n",
         row["id"], row["name"], row["cats"]
end
sth.finish
```

The preceding examples invoke `execute` to get a statement handle, then invoke `finish` when that handle is no longer needed. If instead you invoke `execute` with a code block, it passes the statement handle to the block and invokes `finish` on that handle implicitly:

```
dbh.execute("SELECT id, name, cats FROM profile") do |sth|
  sth.fetch do |row|
    printf "id: %s, name: %s, cats: %s\n", row[0], row[1], row[2]
  end
end
```

Ruby DBI has some high-level database-handle methods for executing statements that produce result sets:

- `select_one` executes a query and returns the first row as an array (or `nil` if the result is empty):

```
row = dbh.select_one("SELECT id, name, cats FROM profile WHERE id = 3")
```

- `select_all` executes a query and returns an array of `DBI::Row` objects, one per row of the result set. The array is empty if the result is empty:

```
rows = dbh.select_all( "SELECT id, name, cats FROM profile")
```

  The `select_all` method is useful when you need to process a result set more than once because Ruby DBI provides no way to "rewind" a result set. By fetching the entire result set as an array of row objects, you can iterate through its elements multiple times. If you need to run through the rows only once, you can apply an iterator directly to `select_all`:

```
dbh.select_all("SELECT id, name, cats FROM profile").each do |row|
  printf "id: %s, name: %s, cats: %s\n",
         row["id"], row["name"], row["cats"]
end
```

PHP

PDO has two connection-object methods to execute SQL statements: `exec()` for statements that do not return a result set and `query()` for those that do. If you have PDO exceptions enabled, both methods raise an exception if statement execution fails. (Another approach couples the `prepare()` and `execute()` methods; see Recipe 2.5.)

To execute statements such as `INSERT` or `UPDATE` that don't return rows, use `exec()`. It returns a count to indicate how many rows were changed:

```
$count = $dbh->exec ("UPDATE profile SET cats = cats+1 WHERE name = 'Sybil'");
printf ("Number of rows updated: %d\n", $count);
```

For statements such as `SELECT` that return a result set, the `query()` method returns a statement handle. Generally, you use this object to call a row-fetching method in a loop, and count the rows if you need to know how many there are:

```
$sth = $dbh->query ("SELECT id, name, cats FROM profile");
$count = 0;
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  printf ("id: %s, name: %s, cats: %s\n", $row[0], $row[1], $row[2]);
  $count++;
}
printf ("Number of rows returned: %d\n", $count);
```

To determine the number of columns in the result set, call the statement handle `columnCount()` method.

The example demonstrates the statement handle `fetch()` method, which returns the next row of the result set or `FALSE` when there are no more. `fetch()` takes an optional argument that indicates what type of value it should return. As shown, with an argument of `PDO::FETCH_NUM`, `fetch()` returns an array with elements accessed using numeric subscripts, beginning with 0. The array size indicates the number of result set columns.

With an argument of `PDO::FETCH_ASSOC`, `fetch()` returns an associative array containing values accessed by column name (`$row["id"]`, `$row["name"]`, `$row["cats"]`).

With an argument of `PDO::FETCH_OBJ`, `fetch()` returns an object having members accessed using the column names (`$row->id`, `$row->name`, `$row->cats`).

`fetch()` uses the default fetch mode if you invoke it with no argument. Unless you have changed the mode, it's `PDO::FETCH_BOTH`, which is like a combination of `PDO::FETCH_NUM` and `PDO::FETCH_ASSOC`. To set the default fetch mode for all statements executed within a connection, use the `setAttribute` database-handle method:

```
$dbh->setAttribute (PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
```

To set the mode for a given statement, call its `setFetchMode()` method after executing the statement and before fetching the results:

```
$sth->setFetchMode (PDO::FETCH_OBJ);
```

It's also possible to use a statement handle as an iterator. The handle uses the current default fetch mode:

```
$sth->setFetchMode (PDO::FETCH_NUM);
foreach ($sth as $row)
  printf ("id: %s, name: %s, cats: %s\n", $row[0], $row[1], $row[2]);
```

The `fetchAll()` method fetches and returns the entire result set as an array of rows. It permits an optional fetch-mode argument:

```
$rows = $sth->fetchAll (PDO::FETCH_NUM);
foreach ($rows as $row)
  printf ("id: %s, name: %s, cats: %s\n", $row[0], $row[1], $row[2]);
```

In this case, the row count is the number of elements in `$rows`.

Python

The Python DB API uses the same calls for SQL statements that do not return a result set and those that do. To process a statement in Python, use your database connection object to get a cursor object. Then use the cursor's `execute()` method to send the statement to the server. If the statement fails with an error, `execute()` raises an exception. Otherwise, if there is no result set, statement execution is complete, and the cursor's `rowcount` attribute indicates how many rows were changed:

```
cursor = conn.cursor()
cursor.execute("UPDATE profile SET cats = cats+1 WHERE name = 'Sybil'")
```

```
print("Number of rows updated: %d" % cursor.rowcount)
cursor.close()
conn.commit()
```

The Python DB API specification indicates that database connections should begin with auto-commit mode *disabled*, so Connector/Python disables auto-commit when it connects to the MySQL server. If you use transactional tables, modifications to them are rolled back when you close the connection unless you commit the changes first, which is why the preceding example invokes the `commit()` method. Changes to nontransactional tables such as MyISAM tables are committed automatically, so this issue does not arise. For more information on auto-commit mode, see Chapter 17, particularly Recipe 17.7).

If the statement returns a result set, fetch its rows, then close the cursor. The `fetchone()` method returns the next row as a sequence, or `None` when there are no more rows:

```
cursor = conn.cursor()
cursor.execute("SELECT id, name, cats FROM profile")
while True:
  row = cursor.fetchone()
  if row is None:
    break
  print("id: %s, name: %s, cats: %s" % (row[0], row[1], row[2]))
print("Number of rows returned: %d" % cursor.rowcount)
cursor.close()
```

As you can see from the preceding example, the `rowcount` attribute is useful for `SELECT` statements, too; it indicates the number of rows in the result set.

`len(row)` tells you the number of columns in the result set.

Alternatively, use the cursor itself as an iterator that returns each row in turn:

```
cursor = conn.cursor()
cursor.execute("SELECT id, name, cats FROM profile")
for (id, name, cats) in cursor:
  print("id: %s, name: %s, cats: %s" % (id, name, cats))
print("Number of rows returned: %d" % cursor.rowcount)
cursor.close()
```

The `fetchall()` method returns the entire result set as a sequence of row sequences. Iterate through the sequence to access the rows:

```
cursor = conn.cursor()
cursor.execute("SELECT id, name, cats FROM profile")
rows = cursor.fetchall()
for row in rows:
  print("id: %s, name: %s, cats: %s" % (row[0], row[1], row[2]))
```

```
print("Number of rows returned: %d" % cursor.rowcount)
cursor.close()
```

DB API provides no way to rewind a result set, so `fetchall()` can be convenient when you must iterate through the rows of the result set more than once or access individual values directly. For example, if `rows` holds the result set, you can access the value of the third column in the second row as `rows[1][2]` (indexes begin at 0, not 1).

Java

The JDBC interface provides specific object types for the various phases of SQL statement processing. Statements are executed in JDBC using Java objects of one type. The results, if any, are returned as objects of another type.

To execute a statement, first get a `Statement` object by calling the `createStatement()` method of your `Connection` object:

```
Statement s = conn.createStatement ();
```

Then use the `Statement` object to send the statement to the server. JDBC provides several methods for doing this. Choose the one that's appropriate for the type of statement: `executeUpdate()` for statements that don't return a result set, `executeQuery()` for statements that do, and `execute()` when you don't know. Each method raises an exception if the statement fails.

The `executeUpdate()` method sends a statement that generates no result set to the server and returns a count indicating the number of affected rows. When you're done with the statement object, close it:

```
Statement s = conn.createStatement ();
int count = s.executeUpdate (
              "UPDATE profile SET cats = cats+1 WHERE name = 'Sybil'");
s.close ();   // close statement
System.out.println ("Number of rows updated: " + count);
```

For statements that return a result set, use `executeQuery()`. Then get a result set object, and use it to retrieve the row values. When you're done, close the result set and statement objects:

```
Statement s = conn.createStatement ();
s.executeQuery ("SELECT id, name, cats FROM profile");
ResultSet rs = s.getResultSet ();
int count = 0;
while (rs.next ())  // loop through rows of result set
{
  int id = rs.getInt (1);   // extract columns 1, 2, and 3
  String name = rs.getString (2);
  int cats = rs.getInt (3);
  System.out.println ("id: " + id
                      + ", name: " + name
                      + ", cats: " + cats);
```

```
  ++count;
}
rs.close ();  // close result set
s.close ();   // close statement
System.out.println ("Number of rows returned: " + count);
```

The `ResultSet` object returned by the `getResultSet()` method of your `Statement` object has its own methods, such as `next()` to fetch rows and various `get`*XXX*`()` methods that access columns of the current row. Initially, the result set is positioned just before the first row of the set. Call `next()` to fetch each row in succession until it returns false. To determine the number of rows in a result set, count them yourself, as shown in the preceding example.

To access column values, use methods such as `getInt()`, `getString()`, `getFloat()`, and `getDate()`. To obtain the column value as a generic object, use `getObject()`. The argument to a `get`*XXX*`()` call can indicate either column position (beginning at 1, not 0) or column name. The previous example shows how to retrieve the `id`, `name`, and `cats` columns by position. To access columns by name instead, write the row-fetching loop as follows:

```
while (rs.next ())  // loop through rows of result set
{
  int id = rs.getInt ("id");
  String name = rs.getString ("name");
  int cats = rs.getInt ("cats");
  System.out.println ("id: " + id
                      + ", name: " + name
                      + ", cats: " + cats);
  ++count;
}
```

To retrieve a given column value, use any `get`*XXX*`()` call that makes sense for the data type. For example, `getString()` retrieves any column value as a string:

```
String id = rs.getString ("id");
String name = rs.getString ("name");
String cats = rs.getString ("cats");
System.out.println ("id: " + id
                    + ", name: " + name
                    + ", cats: " + cats);
```

Or use `getObject()` to retrieve values as generic objects and convert the values as necessary. The following example uses `toString()` to convert object values to printable form:

```
Object id = rs.getObject ("id");
Object name = rs.getObject ("name");
Object cats = rs.getObject ("cats");
System.out.println ("id: " + id.toString ()
                    + ", name: " + name.toString ()
                    + ", cats: " + cats.toString ());
```

To determine the number of columns in the result set, access its metadata:

```
ResultSet rs = s.getResultSet ();
ResultSetMetaData md = rs.getMetaData (); // get result set metadata
int ncols = md.getColumnCount ();         // get column count from metadata
```

The third JDBC statement-execution method, `execute()`, works for either type of statement. It's particularly useful when you receive a statement string from an external source and don't know whether it generates a result set. The return value from `execute()` indicates the statement type so that you can process it appropriately: if `execute()` returns true, there is a result set, otherwise not. Typically, you'd use it something like this, where `stmtStr` represents an arbitrary SQL statement:

```
Statement s = conn.createStatement ();
if (s.execute (stmtStr))
{
  // there is a result set
  ResultSet rs = s.getResultSet ();

  // ... process result set here ...

  rs.close ();  // close result set
}
else
{
  // there is no result set, just print the row count
  System.out.println ("Number of rows affected: " + s.getUpdateCount ());
}
s.close ();   // close statement
```

2.5. Handling Special Characters and NULL Values in Statements

Problem

You need to construct SQL statements that refer to data values containing special characters such as quotes or backslashes, or special values such as `NULL`. Or you are constructing statements using data obtained from external sources and want to prevent SQL injection attacks.

Solution

Use your API's placeholder mechanism or quoting function to make data safe for insertion.

Discussion

Up to this point in the chapter, our statements have used "safe" data values that require no special treatment. For example, we can easily construct the following SQL statements from within a program by writing the data values literally in the statement strings:

```
SELECT * FROM profile WHERE age > 40 AND color = 'green'

INSERT INTO profile (name,color) VALUES('Gary','blue')
```

However, some data values are not so easily handled and cause problems if you are not careful. Statements might use values that contain special characters such as quotes, backslashes, binary data, or values that are `NULL`. The following discussion describes the difficulties these values cause and the proper techniques for handling them.

Suppose that you want to execute this `INSERT` statement:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('Alison','1973-01-12','blue','eggroll',4);
```

There's nothing unusual about that. But if you change the `name` column value to something like `De'Mont` that contains a single quote, the statement becomes syntactically invalid:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De'Mont','1973-01-12','blue','eggroll',4);
```

The problem is the single quote inside a single-quoted string. To make the statement legal by escaping the quote, precede it with either a single quote or a backslash:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De''Mont','1973-01-12','blue','eggroll',4);

INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De\'Mont','1973-01-12','blue','eggroll',4);
```

Alternatively, quote the `name` value itself within double quotes rather than within single quotes (assuming that the `ANSI_QUOTES` SQL mode is not enabled):

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES("De'Mont",'1973-01-12','blue','eggroll',4);
```

If you are writing a statement literally in your program, you can escape or quote the `name` value by hand because you know what the value is. But if the name is stored in a variable, you don't necessarily know what the variable's value is. Worse yet, single quote isn't the only character you must be prepared to deal with; double quotes and backslashes cause problems, too. And if the database stores binary data such as images or sound clips, a value might contain anything—not only quotes or backslashes, but other characters such as nulls (zero-valued bytes). The need to handle special characters properly is particularly acute in a web environment where statements are constructed using form input (for example, if you search for rows that match search terms entered by the remote

user). You must be able to handle any kind of input in a general way because you can't predict in advance what kind of information a user will supply. It is not uncommon for malicious users to enter garbage values containing problematic characters in a deliberate attempt to compromise the security of your server. That is a standard technique for exploiting insecure scripts.

The SQL NULL value is not a special character, but it too requires special treatment. In SQL, NULL indicates "no value." This can have several meanings depending on context, such as "unknown," "missing," "out of range," and so forth. Our statements thus far have not used NULL values, to avoid dealing with the complications that they introduce, but now it's time to address these issues. For example, if you don't know De'Mont's favorite color, you can set the color column to NULL—but not by writing the statement like this:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De''Mont','1973-01-12','NULL','eggroll',4);
```

Instead, the NULL value must have no enclosing quotes:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De''Mont','1973-01-12',NULL,'eggroll',4);
```

Were you writing the statement literally in your program, you'd simply write the word "NULL" without enclosing quotes. But if the color value comes from a variable, the proper action is not so obvious. You must know whether the variable's value represents NULL to determine whether to enclose it within quotes when you construct the statement.

You have two means at your disposal for dealing with special characters such as quotes and backslashes, and with special values such as NULL:

- Use placeholders in the statement string to refer to data values symbolically, then bind the data values to the placeholders when you execute the statement. This is the preferred method because the API itself does all or most of the work for you of providing quotes around values as necessary, quoting or escaping special characters within the data value, and possibly interpreting a special value to map onto NULL without enclosing quotes.
- Use a quoting function (if your API provides one) for converting data values to a safe form that is suitable for use in statement strings.

This section shows how to use these techniques to handle special characters and NULL values for each API. One of the examples demonstrated here shows how to insert a profile table row that contains De'Mont for the name value and NULL for the color value. However, the principles shown here have general utility and handle any special characters, including those found in binary data. (See Chapter 19 for examples showing how to work with images, which are one kind of binary data.) Also, the principles are not limited to INSERT statements. They work for other kinds of statements as well, such as

SELECT. One of the other examples shown here demonstrates how to execute a SELECT statement using placeholders.

Processing of special characters and NULL values comes up in other contexts covered elsewhere:

- The placeholder and quoting techniques described here are *only* for data values and not for identifiers such as database or table names. For discussion of identifier quoting, refer to Recipe 2.6.
- Comparisons of NULL values require different operators than non-NULL values. Recipe 3.6 discusses how to construct SQL statements that perform NULL comparisons from within programs.
- This section covers the issue of getting special characters *into* your database. A related issue is the inverse operation of transforming special characters in values returned *from* your database for display in various contexts. For example, if you generate HTML pages that include values taken from your database, you must perform output encoding to convert < and > characters in those values to the HTML entities < and > to make sure they display properly. Recipe 18.4 discusses that topic.

Using placeholders

Placeholders enable you to avoid writing data values literally in SQL statements. Using this approach, you write statements using placeholders—special markers that indicate where the values go. Two common parameter markers are ? and %s. Depending on the marker, rewrite the INSERT statement to use placeholders like this:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES(?,?,?,?,?)

INSERT INTO profile (name,birth,color,foods,cats)
VALUES(%s,%s,%s,%s,%s)
```

Then pass the statement string to the database server and supply the data values separately. The API binds the values to the placeholders to replace them, resulting in a statement that contains the data values.

One benefit of placeholders is that parameter-binding operations automatically handle escaping of characters such as quotes and backslashes. This is especially useful for inserting binary data such as images into your database or using data values with unknown content such as input submitted by a remote user through a form in a web page. Also, there is usually some special value that you bind to a placeholder to indicate that you want an SQL NULL value in the resulting statement.

A second benefit of placeholders is that you can "prepare" a statement in advance, then reuse it by binding different values to it each time it's executed. Prepared statements thus encourage statement reuse. Statements become more generic because they contain placeholders rather than specific data values. If you perform an operation over and over, you may be able to reuse a prepared statement and simply bind different data values to it each time you execute it. Some database systems (MySQL not among them) have the capability of performing some preparsing or even execution planning prior to executing a prepared statement. For a statement that is executed multiple times later, this reduces overhead because anything that can be done prior to execution need be done only once, not once per execution. For example, if a program executes a particular type of `SELECT` statement several times while it runs, such a database system can construct a plan for the statement and then reuse it each time, rather than rebuild the plan over and over. MySQL doesn't build query plans in advance, so you get no performance boost from using prepared statements. However, if you port a program to a database that does reuse query plans and you've written your program to use prepared statements, you can get this advantage of prepared statements automatically. You need not convert from non-prepared statements to enjoy that benefit.

A third (admittedly subjective) benefit is that code that uses placeholder-based statements can be easier to read. As you work through this section, compare the statements used here with those from Recipe 2.4 that did not use placeholders to see which you prefer.

Using a quoting function

Some APIs provide a quoting function that takes a data value as its argument and returns a properly quoted and escaped value suitable for safe insertion into an SQL statement. This is less common than using placeholders, but it can be useful for constructing statements that you do not intend to execute immediately. However, you must have a connection open to the database server while you use such a quoting function because the API cannot select the proper quoting rules until the database driver is known. (The rules differ among database systems.)

As we'll indicate later, some APIs quote as strings all non-`NULL` values, even numbers, when binding them to parameter markers. This can be an issue in contexts that *require* numbers, as described further in Recipe 3.11.

Generating a List of Placeholders

You cannot bind an array of data values to a single placeholder. Each value must be bound to a separate placeholder. To use placeholders for a list of data values that may

vary in number, construct a list of placeholder characters. In Perl, the following statement creates a string consisting of *n* placeholder characters separated by commas:

```
$str = join (",", ("?") x n);
```

The `x` repetition operator, when applied to a list, produces *n* copies of the list, so the `join()` call joins these lists to produce a single string containing *n* comma-separated instances of the `?` character. This is handy for binding an array of data values to a list of placeholders in a statement string because the size of the array is the number of placeholders needed:

```
$str = join (",", ("?") x @values);
```

In Ruby, use the `*` operator to similar effect:

```
str = (["?"] * values.size).join(",")
```

A less cryptic method is to use a loop approach, here illustrated in Python:

```
str = ""
if len(values) > 0:
  str = "?"
for i in range(1, len(values)):
  str += ",?"
```

Perl

To use placeholders with Perl DBI, put a `?` in your SQL statement string at each data value location. Then bind the values to the statement by passing them to `do()` or `execute()`, or by calling a DBI method specifically intended for placeholder substitution. Use `undef` to bind a `NULL` value to a placeholder.

With `do()`, add the `profile` row for De'Mont by passing the statement string and the data values in the same call:

```
my $count = $dbh->do ("INSERT INTO profile (name,birth,color,foods,cats)
                       VALUES(?,?,?,?,?)",
                      undef,
                      "De'Mont", "1973-01-12", undef, "eggroll", 4);
```

The arguments following the statement string are `undef`, then one data value for each placeholder. The `undef` argument is a historical artifact, but must be present.

Alternatively, pass the statement string to `prepare()` to get a statement handle, then use that handle to pass the data values to `execute()`:

```
my $sth = $dbh->prepare ("INSERT INTO profile (name,birth,color,foods,cats)
                          VALUES(?,?,?,?,?)");
my $count = $sth->execute ("De'Mont", "1973-01-12", undef, "eggroll", 4);
```

In either case, DBI generates this statement:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De\'Mont','1973-01-12',NULL,'eggroll','4')
```

The Perl DBI placeholder mechanism provides quotes around data values when they are bound to the statement string, so don't put quotes around the ? characters in the string.

Note that the placeholder mechanism adds quotes around numeric values. DBI relies on the MySQL server to perform type conversion as necessary to convert strings to numbers. If you bind `undef` to a placeholder, DBI puts a `NULL` into the statement and correctly refrains from adding enclosing quotes.

To execute the same statement over and over again, use `prepare()` once, then call `execute()` with appropriate data values each time you run it.

You can use these methods for other types of statements as well. For example, the following `SELECT` statement uses a placeholder to look for rows that have a `cats` value larger than 2:

```
my $sth = $dbh->prepare ("SELECT * FROM profile WHERE cats > ?");
$sth->execute (2);
while (my $ref = $sth->fetchrow_hashref ())
{
  print "id: $ref->{id}, name: $ref->{name}, cats: $ref->{cats}\n";
}
```

High-level retrieval methods such as `selectrow_array()` and `selectall_arrayref()` can be used with placeholders, too. Like the `do()` method, the arguments are the statement string, `undef`, and the data values to bind to the placeholders. Here's an example:

```
my $ref = $dbh->selectall_arrayref (
  "SELECT name, birth, foods FROM profile WHERE id > ? AND color = ?",
  undef, 3, "green"
);
```

The Perl DBI `quote()` database- handle method is an alternative to using placeholders. Here's how to use `quote()` to create a statement string that inserts a new row in the `profile` table. Write the `%s` format specifiers without enclosing quotes because `quote()` provides them automatically as necessary. Non-`undef` values are inserted with quotes, and `undef` values are inserted as `NULL` without quotes:

```
my $stmt = sprintf ("INSERT INTO profile (name,birth,color,foods,cats)
                     VALUES(%s,%s,%s,%s,%s)",
                    $dbh->quote ("De'Mont"),
                    $dbh->quote ("1973-01-12"),
                    $dbh->quote (undef),
                    $dbh->quote ("eggroll"),
                    $dbh->quote (4));
my $count = $dbh->do ($stmt);
```

The statement string generated by this code is the same as when you use placeholders.

Ruby

Ruby DBI uses `?` as the placeholder character in SQL statements and `nil` as the value for binding an SQL `NULL` value to a placeholder.

To use placeholders with `do`, pass the statement string followed by the data values to bind to the placeholders:

```
count = dbh.do("INSERT INTO profile (name,birth,color,foods,cats)
                VALUES(?,?,?,?,?)",
               "De'Mont", "1973-01-12", nil, "eggroll", 4)
```

Alternatively, pass the statement string to `prepare` to get a statement handle, then use that handle to invoke `execute` with the data values:

```
sth = dbh.prepare("INSERT INTO profile (name,birth,color,foods,cats)
                   VALUES(?,?,?,?,?)")
count = sth.execute("De'Mont", "1973-01-12", nil, "eggroll", 4)
```

Regardless of how you construct the statement, DBI includes properly escaped quotes and a properly unquoted `NULL` value:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De\'Mont','1973-01-12',NULL,'eggroll',4)
```

The Ruby DBI placeholder mechanism provides quotes around data values as necessary when they are bound to the statement string, so don't put quotes around the `?` characters in the string.

The approach that uses `prepare` plus `execute` is useful for a statement to be executed multiple times with different data values. For a statement to be executed just once, you can skip the `prepare` step. Pass the statement string and the data values to the database handle `execute` method:

```
sth = dbh.execute("SELECT * FROM profile WHERE cats > ?", 2)
sth.fetch do |row|
  printf "id: %s, name: %s, cats: %s\n", row["id"], row["name"], row["cats"]
end
sth.finish
```

The Ruby DBI `quote()` database-handle method is an alternative to placeholders. The following example uses `quote()` to produce the `INSERT` statement for De'Mont. Write the `%s` format specifiers without enclosing quotes because `quote()` provides them automatically as necessary. Non-`nil` values are inserted with quotes, and `nil` values are inserted as `NULL` without quotes:

```
stmt = sprintf "INSERT INTO profile (name,birth,color,foods,cats)
                VALUES(%s,%s,%s,%s,%s)",
               dbh.quote("De'Mont"),
               dbh.quote("1973-01-12"),
```

```
                dbh.quote(nil),
                dbh.quote("eggroll"),
                dbh.quote(4)
count = dbh.do(stmt)
```

The statement string generated by this code is the same as when you use placeholders.

PHP

To use placeholders with the PDO extension, pass a statement string to `prepare()` to get a statement object. The string can contain `?` characters as placeholder markers. Use this object to invoke `execute()`, passing to it the array of data values to bind to the placeholders. Use the PHP `NULL` value to bind an SQL `NULL` value to a placeholder. The code to add the `profile` table row for De'Mont looks like this:

```
$sth = $dbh->prepare ("INSERT INTO profile (name,birth,color,foods,cats)
                       VALUES(?,?,?,?,?)");
$sth->execute (array ("De'Mont","1973-01-12",NULL,"eggroll",4));
```

The resulting statement includes a properly escaped quote and a properly unquoted `NULL` value:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De\'Mont','1973-01-12',NULL,'eggroll','4')
```

The PDO placeholder mechanism provides quotes around data values when they are bound to the statement string, so don't put quotes around the `?` characters in the string. (Note that even the numeric value `4` is quoted; PDO relies on MySQL to perform type conversion as necessary when the statement executes.)

Python

The Connector/Python module implements placeholders using `%s` format specifiers in the SQL statement string. (To place a literal `%` character into the statement, use `%%` in the statement string.) To use placeholders, invoke the `execute()` method with two arguments: a statement string containing format specifiers and a sequence containing the values to bind to the statement string. Use `None` to bind a `NULL` value to a placeholder. The code to add the `profile` table row for De'Mont looks like this:

```
cursor = conn.cursor()
cursor.execute('''
               INSERT INTO profile (name,birth,color,foods,cats)
               VALUES(%s,%s,%s,%s,%s)
               ''', ("De'Mont", "1973-01-12", None, "eggroll", 4))
cursor.close()
conn.commit()
```

The statement sent to the server by the preceding `execute()` call looks like this:

```
INSERT INTO profile (name,birth,color,foods,cats)
VALUES('De\'Mont','1973-01-12',NULL,'eggroll',4)
```

The Connector/Python placeholder mechanism provides quotes around data values as necessary when they are bound to the statement string, so don't put quotes around the `%s` format specifiers in the string.

If you have only a single value *`val`* to bind to a placeholder, write it as a sequence using the syntax `(`*`val`*`,)`:

```
cursor = conn.cursor()
cursor.execute("SELECT id, name, cats FROM profile WHERE cats = %s", (2,))
for (id, name, cats) in cursor:
  print("id: %s, name: %s, cats: %s" % (id, name, cats))
cursor.close()
```

Alternatively, write the value as a list using the syntax `[`*`val`*`]`.

Java

JDBC provides support for placeholders if you use prepared statements. Recall that the process for executing nonprepared statements in JDBC is to create a `Statement` object, and then pass the statement string to the `executeUpdate()`, `executeQuery()`, or `execute()` function. To use a prepared statement instead, create a `PreparedStatement` object by passing a statement string containing `?` placeholder characters to your connection object's `prepareStatement()` method. Then bind the data values to the statement using `set`*`XXX`*`()` methods. Finally, execute the statement by calling `executeUpdate()`, `executeQuery()`, or `execute()` with an empty argument list.

Here is an example that uses `executeUpdate()` to execute an `INSERT` statement that adds the `profile` table row for De'Mont:

```
PreparedStatement s;
s = conn.prepareStatement (
            "INSERT INTO profile (name,birth,color,foods,cats)"
            + " VALUES(?,?,?,?,?)");
s.setString (1, "De'Mont");         // bind values to placeholders
s.setString (2, "1973-01-12");
s.setNull (3, java.sql.Types.CHAR);
s.setString (4, "eggroll");
s.setInt (5, 4);
s.close ();   // close statement
```

The `set`*`XXX`*`()` methods that bind data values to statements take two arguments: a placeholder position (beginning with 1, not 0) and the value to bind to the placeholder. Choose each value-binding call to match the data type of the column to which the value is bound: `setString()` to bind a string to the `name` column, `setInt()` to bind an integer to the `cats` column, and so forth. (Actually, I cheated a bit by using `setString()` to treat the date value for `birth` as a string.)

One difference between JDBC and the other APIs is that you don't bind a `NULL` to a placeholder by specifying some special value (such as `undef` in Perl or `nil` in Ruby).

Instead, invoke `setNull()` with a second argument that indicates the type of the column: `java.sql.Types.CHAR` for a string, `java.sql.Types.INTEGER` for an integer, and so forth.

The `set`*`XXX`*`()` calls add quotes around data values if necessary, so don't put quotes around the `?` placeholder characters in the statement string.

To handle a statement that returns a result set, the process is similar, but execute the prepared statement with `executeQuery()` rather than `executeUpdate()`:

```
PreparedStatement s;
s = conn.prepareStatement ("SELECT * FROM profile WHERE cats > ?");
s.setInt (1, 2);  // bind 2 to first placeholder
s.executeQuery ();
// ... process result set here ...
s.close ();     // close statement
```

2.6. Handling Special Characters in Identifiers

Problem

You need to construct SQL statements that refer to identifiers containing special characters.

Solution

Quote each identifier so it can be inserted safely into statement strings.

Discussion

Recipe 2.5 discusses how to handle special characters in data values by using placeholders or quoting methods. Special characters also can be present in identifiers such as database, table, and column names. For example, the table name `some table` contains a space, which is not permitted by default:

```
mysql> CREATE TABLE some table (i INT);
ERROR 1064 (42000): You have an error in your SQL syntax near 'table (i INT)'
```

Special characters are handled differently in identifiers than in data values. To make an identifier safe for insertion into an SQL statement, quote it by enclosing it within backticks:

```
mysql> CREATE TABLE `some table` (i INT);
Query OK, 0 rows affected (0.04 sec)
```

In MySQL, backticks are always permitted for identifier quoting. The double-quote character is permitted as well, if the `ANSI_QUOTES` SQL mode is enabled. Thus, with `ANSI_QUOTES` enabled, both of these statements are equivalent:

```
CREATE TABLE `some table` (i INT);
CREATE TABLE "some table" (i INT);
```

If it's necessary to know which identifier quoting characters are permitted, execute a `SELECT @@sql_mode` statement to retrieve the SQL mode and check whether its value includes `ANSI_QUOTES`.

If a quoting character appears within the identifier itself, double it when quoting the identifier. For example, quote `abc`def` as `` `abc``def` ``.

Be aware that although string data values in MySQL normally can be quoted using either single-quote or double-quote characters (`'abc'`, `"abc"`), that is not true when `ANSI_QUOTES` is enabled. In that case, MySQL interprets `'abc'` as a string and `"abc"` as an identifier, so you must use only single quotes for strings.

Within a program, you can use an identifier-quoting routine if your API provides one, or write one yourself if not. Perl DBI has a `quote_identifier()` method that returns a properly quoted identifier. For an API that has no such method, you can quote an identifier by enclosing it within backticks and doubling any backticks that occur within the identifier. Here's a PHP routine that does so:

```
function quote_identifier ($ident)
{
  return ('`' . str_replace('`', '``', $ident) . '`');
}
```

Portability note: If you write your own identifier-quoting routines, remember that other DBMSs may require different quoting conventions.

In contexts where identifiers are used as data values, handle them as such. If you select information from the `INFORMATION_SCHEMA` metadata database, it's common to indicate which rows to return by specifying database object names in the `WHERE` clause. For example, this statement retrieves the column names for the `profile` table in the `cookbook` database:

```
SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'profile';
```

The database and table names are used here as data values, not as identifiers. Were you to construct this statement within a program, parameterize them using placeholders, not identifier quoting. For example, in Ruby, do this:

```
names = dbh.select_all("SELECT COLUMN_NAME
                        FROM INFORMATION_SCHEMA.COLUMNS
                        WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?",
                       db_name, tbl_name)
```

2.7. Identifying NULL Values in Result Sets

Problem

A query result includes NULL values, but you're not sure how to identify them.

Solution

Your API probably has some special value that represents NULL by convention. You just have to know what it is and how to test for it.

Discussion

Recipe 2.5 describes how to refer to NULL values when you send statements *to* the database server. In this section, we'll deal instead with the question of how to recognize and process NULL values returned *from* the database server. In general, this is a matter of knowing what special value the API maps NULL values to, or what method to call. The following table shows these values:

Language	NULL-detection value or method
Perl DBI	`undef` value
Ruby DBI	`nil` value
PHP PDO	`NULL` value
Python DB API	`None` value
Java JDBC	`wasNull()` method

The following sections show a very simple application of NULL value detection. The examples retrieve a result set and print all values in it, mapping NULL values onto the printable string "NULL".

To make sure that the `profile` table has a row that contains some NULL values, use *mysql* to execute the following INSERT statement, then execute the SELECT statement to verify that the resulting row has the expected values:

```
mysql> INSERT INTO profile (name) VALUES('Amabel');
mysql> SELECT * FROM profile WHERE name = 'Amabel';
+----+--------+-------+-------+-------+------+
| id | name   | birth | color | foods | cats |
+----+--------+-------+-------+-------+------+
|  9 | Amabel | NULL  | NULL  | NULL  | NULL |
+----+--------+-------+-------+-------+------+
```

The `id` column might contain a different number, but the other columns should appear as shown, with values of NULL.

Perl

Perl DBI represents NULL values using undef. To detect such values, use the defined() function; it's particularly important to do so if you enable warnings with the Perl -w option or by including a use warnings line in your script. Otherwise, accessing undef values causes Perl to issue Use of uninitialized value warnings.

To prevent these warnings, test column values that might be undef with defined() before using them. The following code selects a few columns from the profile table and prints "NULL" for any undefined values in each row. This makes NULL values explicit in the output without activating any warning messages:

```
my $sth = $dbh->prepare ("SELECT name, birth, foods FROM profile");
$sth->execute ();
while (my $ref = $sth->fetchrow_hashref ())
{
  printf "name: %s, birth: %s, foods: %s\n",
         defined ($ref->{name}) ? $ref->{name} : "NULL",
         defined ($ref->{birth}) ? $ref->{birth} : "NULL",
         defined ($ref->{foods}) ? $ref->{foods} : "NULL";
}
```

Unfortunately, testing multiple column values is ponderous and becomes worse the more columns there are. To avoid this, test and set undefined values using a loop or map prior to printing them. The following example uses map:

```
my $sth = $dbh->prepare ("SELECT name, birth, foods FROM profile");
$sth->execute ();
while (my $ref = $sth->fetchrow_hashref ())
{
  map { $ref->{$_} = "NULL" unless defined ($ref->{$_}); } keys (%{$ref});
  printf "name: %s, birth: %s, foods: %s\n",
         $ref->{name}, $ref->{birth}, $ref->{foods};
}
```

With this technique, the amount of code to perform the tests is constant, not proportional to the number of columns to be tested. Also, there is no reference to specific column names, so it can more easily be used in other programs or as the basis for a utility routine.

If you fetch rows into an array rather than into a hash, use map like this to convert undef values:

```
my $sth = $dbh->prepare ("SELECT name, birth, foods FROM profile");
$sth->execute ();
while (my @val = $sth->fetchrow_array ())
{
  @val = map { defined ($_) ? $_ : "NULL" } @val;
  printf "name: %s, birth: %s, foods: %s\n",
         $val[0], $val[1], $val[2];
}
```

Ruby

Ruby DBI represents NULL values using `nil`, which can be identified by applying the `nil?` method to a value. The following example uses `nil?` to determine whether to print result set values as is or as the string "NULL" for NULL values:

```
dbh.execute("SELECT name, birth, foods FROM profile") do |sth|
  sth.fetch do |row|
    for i in 0...row.length
      row[i] = "NULL" if row[i].nil?  # is the column value NULL?
    end
    printf "id: %s, name: %s, cats: %s\n", row[0], row[1], row[2]
  end
end
```

A shorter alternative to the `for` loop is the `collect!` method, which takes each array element in turn and replaces it with the value returned by the code block:

```
row.collect! { |val| val.nil? ? "NULL" : val }
```

PHP

PHP represents SQL NULL values in result sets as the PHP NULL value. To determine whether a value from a result set represents a NULL value, compare it to the PHP NULL value using the === "triple equal" operator:

```
if ($val === NULL)
{
  # $val is a NULL value
}
```

In PHP, the triple equal operator means "exactly equal to." The usual == "equal to" comparison operator is not suitable here: with ==, PHP considers the NULL value, the empty string, and `0` all equal.

The following code uses the === operator to identify NULL values in a result set and print them as the string "NULL":

```
$sth = $dbh->query ("SELECT name, birth, foods FROM profile");
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  foreach (array_keys ($row) as $key)
  {
    if ($row[$key] === NULL)
      $row[$key] = "NULL";
  }
  print ("name: $row[0], birth: $row[1], foods: $row[2]\n");
}
```

An alternative to === for NULL value tests is `is_null()`.

Python

Python DB API programs represent NULL in result sets using None. The following example shows how to detect NULL values:

```
cursor = conn.cursor()
cursor.execute("SELECT name, birth, foods FROM profile")
for row in cursor:
  row = list(row)  # convert nonmutable tuple to mutable list
  for i, value in enumerate(row):
    if value is None:  # is the column value NULL?
      row[i] = "NULL"
  print("name: %s, birth: %s, foods: %s" % (row[0], row[1], row[2]))
cursor.close()
```

The inner loop checks for NULL column values by looking for None and converts them to the string "NULL". The example converts row to a mutable object prior to the loop because fetchall() returns rows as sequence values, which are nonmutable (read only).

Java

For JDBC programs, if it's possible for a column in a result set to contain NULL values, it's best to check for them explicitly. The way to do this is to fetch the value and then invoke wasNull(), which returns true if the column is NULL and false otherwise. For example:

```
Object obj = rs.getObject (index);
if (rs.wasNull ())
{ /* the value's a NULL */ }
```

The preceding example uses getObject(), but the principle holds for other get*XXX*() calls as well.

Here's an example that prints each row of a result set as a comma-separated list of values, with "NULL" printed for each NULL value:

```
Statement s = conn.createStatement ();
s.executeQuery ("SELECT name, birth, foods FROM profile");
ResultSet rs = s.getResultSet ();
ResultSetMetaData md = rs.getMetaData ();
int ncols = md.getColumnCount ();
while (rs.next ())  // loop through rows of result set
{
  for (int i = 0; i < ncols; i++) // loop through columns
  {
    String val = rs.getString (i+1);
    if (i > 0)
      System.out.print (", ");
    if (rs.wasNull ())
      System.out.print ("NULL");
    else
      System.out.print (val);
```

```
    }
    System.out.println ();
  }
  rs.close ();  // close result set
  s.close ();   // close statement
```

2.8. Techniques for Obtaining Connection Parameters

Problem

You need to obtain connection parameters for a script so that it can connect to a MySQL server.

Solution

There are several ways to do this. Take your pick from the alternatives described here.

Discussion

Any program that connects to MySQL specifies connection parameters such as the username, password, and hostname. The recipes shown so far have put connection parameters directly into the code that attempts to establish the connection, but that is not the only way for your programs to obtain the parameters. This discussion briefly surveys some of the available techniques:

Hardwire the parameters into the program
: The parameters can be given either in the main source file or in a library file used by the program. This technique is convenient because users need not enter the values themselves, but it's also inflexible. To change parameters, you must modify your program.

Ask for the parameters interactively
: In a command-line environment, you can ask the user a series of questions. In a web or GUI environment, you might do this by presenting a form or dialog. Either way, this becomes tedious for people who use the application frequently, due to the need to enter the parameters each time.

Get the parameters from the command line
: You can use this method either for commands run interactively or from within a script. Like the method of obtaining parameters interactively, you must supply parameters for each command invocation. (A factor that mitigates this burden is that many shells enable you to easily recall commands from your history list for reexecution.)

Get the parameters from the execution environment
: The most common way to do this is to set the appropriate environment variables in one of your shell's startup files (such as *.profile* for *sh*, *bash*, *ksh*; or *.login* for *csh* or *tcsh*). Programs that you run during your login session then can get parameter values by examining their environment.

Get the parameters from a separate file
: With this method, store information such as the username and password in a file that programs can read before connecting to the MySQL server. Reading parameters from a file that's separate from your program gives you the benefit of not having to enter them each time you use the program, without hardwiring the values into it. Also, storing the values in a file enables you to centralize parameters for use by multiple programs, and for security purposes you can set the file access mode to keep other users from reading the file.

  The MySQL client library itself supports an option file mechanism, although not all APIs provide access to it. For those that don't, workarounds may exist. (As an example, Java supports the use of properties files and supplies utility routines for reading them.)

Use a combination of methods
: It's often useful to combine methods, to give users the flexibility of providing parameters different ways. For example, MySQL clients such as *mysql* and *mysqladmin* look for option files in several locations and read any that are present. They then check the command-line arguments for further parameters. This enables users to specify connection parameters in an option file or on the command line.

These methods of obtaining connection parameters do involve security issues:

- Any method that stores connection parameters in a file may compromise your system's security unless the file is protected against access by unauthorized users. This is true whether parameters are stored in a source file, an option file, or a script that invokes a command and specifies the parameters on the command line. (Web scripts that can be read only by the web server don't qualify as secure if other users have administrative access to the server.)
- Parameters specified on the command line or in environment variables are not particularly secure. While a program is executing, its command-line arguments and environment may be visible to other users who run process status commands such as *ps* `-e`. In particular, storing the password in an environment variable perhaps is best limited to those situations in which you're the only user on the machine or you trust all other users.

The rest of this section discusses how to process command-line arguments to get connection parameters and how to read parameters from option files.

Getting parameters from the command line

The convention used by standard clients such as *mysql* and *mysqladmin* for command-line arguments is to permit parameters to be specified using either a short option or a long option. For example, the username `cbuser` can be specified either as `-u cbuser` (or `-ucbuser`) or `--user=cbuser`. In addition, for either of the password options (`-p` or `--password`), the password value may be omitted after the option name to cause the program to prompt for the password interactively.

The standard flags for these command options are `-h` or `--host`, `-u` or `--user`, and `-p` or `--password`. You could write your own code to iterate through the argument list, but it's much easier to use existing option-processing modules written for that purpose. Under the *api* directory of the `recipes` distribution, you'll find example programs that show how to process command arguments to obtain the hostname, username, and password for Perl, Ruby, Python, and Java. An accompanying PDF file explains how each one works.

Insofar as possible, the programs mimic option-handling behavior of the standard MySQL clients. An exception is that option-processing libraries may not permit making the password value optional, and they provide no way of prompting the user for a password interactively if a password option is specified without a password value. Consequently, the programs are written so that if you use `-p` or `--password`, you must provide the password value following the option.

Getting parameters from option files

If your API supports it, you can specify connection parameters in a MySQL option file and let the API read the parameters from the file for you. For APIs that do not support option files directly, you may be able to arrange to read other types of files in which parameters are stored or to write your own functions that read option files.

Recipe 1.4 describes the format of MySQL option files. I assume that you've read the discussion there and concentrate here on how to use option files from within programs. You can find files containing the code discussed here under the *api* directory of the `recipes` distribution.

Under Unix, user-specific options are specified by convention in *~/.my.cnf* (that is, in the *.my.cnf* file in your home directory). However, the MySQL option-file mechanism can look in several different files if they exist, although no option file is *required* to exist. (For the list of standard locations in which MySQL programs look for them, see Recipe 1.4.) If multiple option files exist and a given parameter is specified in several of them, the last value found takes precedence.

Programs you write do not use MySQL option files unless you tell them to:

- Perl DBI and Ruby DBI provide direct API support for reading option files; simply indicate that you want to use them at the time that you connect to the server. It's possible to specify that only a particular file should be read, or that the standard search order should be used to look for multiple option files.
- PHP PDO, Connector/Python, and Java do not support option files. (The PDO MySQL driver does, but not if you use `mysqlnd` as the underlying library.) As a workaround for PHP, we'll write a simple option-file parsing function. For Java, we'll adopt a different approach that uses properties files.

Although the conventional name under Unix for the user-specific option file is *.my.cnf* in the current user's home directory, there's no rule that your own programs must use this particular file. You can name an option file anything you like and put it wherever you want. For example, you might set up a file named *mcb.cnf* and install it in the */usr/local/lib/mcb* directory for use by scripts that access the `cookbook` database. Under some circumstances, you might even want to create multiple option files. Then, from within any given script, select the file that's appropriate for the access privileges the script needs. For example, you might have one option file, *mcb.cnf*, that lists parameters for a full-access MySQL account, and another file, *mcb-readonly.cnf*, that lists connection parameters for an account that needs only read-only access to MySQL. Another possibility is to list multiple groups within the same option file and have your scripts select options from the appropriate group.

Perl. Perl DBI scripts can use option files. To take advantage of this, place the appropriate option specifiers in the third component of the data source name (DSN) string:

- To specify an option group, use `mysql_read_default_group=`*`groupname`*. This tells MySQL to search the standard option files for options in the named group and in the `[client]` group. Write the *`groupname`* value without the surrounding square brackets. (If a group in an option file begins with a `[my_prog]` line, specify the *`groupname`* value as `my_prog`.) To search the standard files but look only in the `[client]` group, *`groupname`* should be `client`.
- To name a specific option file, use `mysql_read_default_file=`*`filename`* in the DSN. When you do this, MySQL looks only in that file and only for options in the `[client]` group.
- If you specify both an option file and an option group, MySQL reads only the named file, but looks for options both in the named group and in the `[client]` group.

The following example tells MySQL to use the standard option-file search order to look for options in both the `[cookbook]` and `[client]` groups:

```
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
# basic DSN
```

```
my $dsn = "DBI:mysql:database=cookbook";
# look in standard option files; use [cookbook] and [client] groups
$dsn .= ";mysql_read_default_group=cookbook";
my $dbh = DBI->connect ($dsn, undef, undef, $conn_attrs);
```

The next example explicitly names the option file located in `$ENV{HOME}`, the home directory of the user running the script. Thus, MySQL looks only in that file and uses options from the `[client]` group:

```
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
# basic DSN
my $dsn = "DBI:mysql:database=cookbook";
# look in user-specific option file owned by the current user
$dsn .= ";mysql_read_default_file=$ENV{HOME}/.my.cnf";
my $dbh = DBI->connect ($dsn, undef, undef, $conn_attrs);
```

If you pass an empty value (`undef` or the empty string) for the username or password arguments of the `connect()` call, `connect()` uses whatever values are found in the option file or files. A nonempty username or password in the `connect()` call overrides any option-file value. Similarly, a host named in the DSN overrides any option-file value. Use this behavior to enable DBI scripts to obtain connection parameters both from option files as well as from the command line as follows:

1. Create `$host_name`, `$user_name`, and `$password` variables, each with a value of `undef`. Then parse the command-line arguments to set the variables to non-`undef` values if the corresponding options are present on the command line. (The *cmdline.pl* Perl script under the *api* directory of the `recipes` distribution demonstrates how to do this.)
2. After parsing the command arguments, construct the DSN string, and call `connect()`. Use `mysql_read_default_group` and `mysql_read_default_file` in the DSN to specify how you want option files to be used, and, if `$host_name` is not `undef`, add `host=$host_name` to the DSN. In addition, pass `$user_name` and `$password` as the username and password arguments to `connect()`. These will be `undef` by default; if they were set from the command-line arguments, they will have non-`undef` values that override any option-file values.

If a script follows this procedure, parameters given by the user on the command line are passed to `connect()` and take precedence over the contents of option files.

Ruby. Ruby DBI scripts can access option files by using a mechanism analogous to that used for Perl DBI, and the following examples correspond exactly to those shown in the preceding Perl discussion.

This example uses the standard option-file search order to look for options in both the `[cookbook]` and `[client]` groups:

```
# basic DSN
dsn = "DBI:Mysql:database=cookbook"
# look in standard option files; use [cookbook] and [client] groups
dsn << ";mysql_read_default_group=cookbook"
dbh = DBI.connect(dsn, nil, nil)
```

The following example uses the *.my.cnf* file in the current user's home directory to obtain parameters from the `[client]` group:

```
# basic DSN
dsn = "DBI:Mysql:database=cookbook"
# look in user-specific option file owned by the current user
dsn << ";mysql_read_default_file=#{ENV['HOME']}/.my.cnf"
dbh = DBI.connect(dsn, nil, nil)
```

PHP. As mentioned earlier, the PDO MySQL driver does not necessarily support using MySQL option files (it does not if you use `mysqlnd` as the underlying library). To work around that limitation, use a function that reads an option file, such as the `read_mysql_option_file()` function shown in the following listing. It takes as arguments the name of an option file and an option group name or an array containing group names. (Group names should be written without square brackets.) It then reads any options present in the file for the named group or groups. If no option group argument is given, the function looks by default in the `[client]` group. The return value is an array of option name/value pairs, or `FALSE` if an error occurs. It is not an error for the file not to exist. (Note that quoted option values and trailing `#`-style comments following option values are legal in MySQL option files, but this function does not handle those constructs.)

```
function read_mysql_option_file ($filename, $group_list = "client")
{
  if (is_string ($group_list))          # convert string to array
    $group_list = array ($group_list);
  if (!is_array ($group_list))          # hmm ... garbage argument?
    return (FALSE);
  $opt = array ();                      # option name/value array
  if (!@($fp = fopen ($filename, "r")))  # if file does not exist,
    return ($opt);                      # return an empty list
  $in_named_group = 0;  # set nonzero while processing a named group
  while ($s = fgets ($fp, 1024))
  {
    $s = trim ($s);
    if (preg_match ("/^[#;]/", $s))             # skip comments
      continue;
    if (preg_match ("/^\[([^]]+)]/", $s, $arg))  # option group line?
    {
      # check whether we are in one of the desired groups
      $in_named_group = 0;
      foreach ($group_list as $group_name)
      {
        if ($arg[1] == $group_name)
```

```
        {
          $in_named_group = 1;    # we are in a desired group
          break;
        }
      }
      continue;
    }
    if (!$in_named_group)         # we are not in a desired
      continue;                   # group, skip the line
    if (preg_match ("/^([^ \t=]+)[ \t]*=[ \t]*(.*)/", $s, $arg))
      $opt[$arg[1]] = $arg[2];    # name=value
    else if (preg_match ("/^([^ \t]+)/", $s, $arg))
      $opt[$arg[1]] = "";         # name only
    # else line is malformed
  }
  return ($opt);
}
```

Here are two examples showing how to use `read_mysql_option_file()`. The first reads a user's option file to get the `[client]` group parameters and uses them to connect to the server. The second reads the system-wide option file, */etc/my.cnf*, and prints the server startup parameters that are found there (that is, the parameters in the `[mysqld]` and `[server]` groups):

```
$opt = read_mysql_option_file ("/home/paul/.my.cnf");
$dsn = "mysql:dbname=cookbook";
if (isset ($opt["host"]))
  $dsn .= ";host=" . $opt["host"];
$user = $opt["user"];
$password = $opt["password"];
try
{
  $dbh = new PDO ($dsn, $user, $password);
  print ("Connected\n");
  $dbh = NULL;
  print ("Disconnected\n");
}
catch (PDOException $e)
{
  print ("Cannot connect to server\n");
}

$opt = read_mysql_option_file ("/etc/my.cnf", array ("mysqld", "server"));
foreach ($opt as $name => $value)
  print ("$name => $value\n");
```

PHP does have a `parse_ini_file()` function that is intended for parsing *.ini* files. These have a syntax that is similar to MySQL option files, so you might find this function of use. However, there are some differences to watch out for. Suppose that you have a file written like this:

```
[client]
user=paul

[client]
host=127.0.0.1

[mysql]
no-auto-rehash
```

Standard MySQL option parsing considers both the `user` and `host` values part of the `[client]` group, whereas `parse_ini_file()` returns only the contents of the final `[client]` stanza; the `user` option is lost. Also, `parse_ini_file()` ignores options that are given without a value, so the `no-auto-rehash` option is lost.

Java. The JDBC MySQL Connector/J driver doesn't support option files. However, the Java class library supports reading properties files that contain lines in *`name=value`* format. This is similar but not identical to MySQL option-file format (for example, properties files do not permit [*`groupname`*] lines). Here is a simple properties file:

```
# this file lists parameters for connecting to the MySQL server
user=cbuser
password=cbpass
host=localhost
```

The following program, *ReadPropsFile.java*, shows one way to read a properties file named *Cookbook.properties* to obtain connection parameters. The file must be in some directory named in your `CLASSPATH` variable, or you must specify it using a full pathname (the example shown here assumes that the file is in a `CLASSPATH` directory):

```
import java.sql.*;
import java.util.*;   // need this for properties file support

public class ReadPropsFile
{
  public static void main (String[] args)
  {
    Connection conn = null;
    String url = null;
    String propsFile = "Cookbook.properties";
    Properties props = new Properties ();

    try
    {
      props.load (ReadPropsFile.class.getResourceAsStream (propsFile));
    }
    catch (Exception e)
    {
      System.err.println ("Cannot read properties file");
      System.exit (1);
    }
    try
```

```
    {
      // construct connection URL, encoding username
      // and password as parameters at the end
      url = "jdbc:mysql://"
            + props.getProperty ("host")
            + "/cookbook"
            + "?user=" + props.getProperty ("user")
            + "&password=" + props.getProperty ("password");
      Class.forName ("com.mysql.jdbc.Driver").newInstance ();
      conn = DriverManager.getConnection (url);
      System.out.println ("Connected");
    }
    catch (Exception e)
    {
      System.err.println ("Cannot connect to server");
    }
    finally
    {
      try
      {
        if (conn != null)
        {
          conn.close ();
          System.out.println ("Disconnected");
        }
      }
      catch (SQLException e) { /* ignore close errors */ }
    }
  }
}
```

To have `getProperty()` return a particular default value when the named property is not found, pass that value as a second argument. For example, to use `127.0.0.1` as the default `host` value, call `getProperty()` like this:

```
String hostName = props.getProperty ("host", "127.0.0.1");
```

The *Cookbook.java* library file developed elsewhere in the chapter (see Recipe 2.3) includes an extra library call in the version of the file that you'll find in the *lib* directory of the `recipes` distribution: a `propsConnect()` routine that is based on the concepts discussed here. To use it, set up the contents of the properties file, *Cookbook.properties*, and copy the file to the same location where you installed *Cookbook.class*. You can then establish a connection within a program by importing the `Cookbook` class and calling `Cookbook.propsConnect()` rather than by calling `Cookbook.connect()`.

2.9. Conclusion and Words of Advice

This chapter discussed the basic operations provided by each of our APIs for handling various aspects of interaction with the MySQL server. These operations enable you to write programs that execute any kind of statement and retrieve the results. Up to this

point, we've used simple statements because the focus is on the APIs rather than on SQL. The next chapter focuses on SQL instead, to show how to ask the database server more complex questions.

Before you proceed, it's a good idea to reset the `profile` table used in this chapter to a known state. Change location into the *tables* directory of the `recipes` distribution, and run these commands:

```
% mysql cookbook < profile.sql
% mysql cookbook < profile2.sql
```

Several statements in later chapters use the `profile` table; by reinitializing it, you'll get the same results displayed in those chapters when you run the statements shown there.

CHAPTER 3

Selecting Data from Tables

3.0. Introduction

This chapter focuses on using the SELECT statement to retrieve information from your database. You will find the chapter helpful if your SQL background is limited or to find out about the MySQL-specific extensions to SELECT syntax.

There are many ways to write SELECT statements; we'll look at only a few. Consult the *MySQL Reference Manual* or a general MySQL text for more information about SELECT syntax and the functions and operators available to extract and manipulate data.

Many examples in this chapter use a table named `mail` that contains rows that track mail message traffic between users on a set of hosts:

```
CREATE TABLE mail
(
  t       DATETIME,    # when message was sent
  srcuser VARCHAR(8),  # sender (source user and host)
  srchost VARCHAR(20),
  dstuser VARCHAR(8),  # recipient (destination user and host)
  dsthost VARCHAR(20),
  size    BIGINT,      # message size in bytes
  INDEX (t)
);
```

The `mail` table contents look like this:

```
mysql> SELECT * FROM mail;
+---------------------+---------+---------+---------+---------+---------+
| t                   | srcuser | srchost | dstuser | dsthost | size    |
+---------------------+---------+---------+---------+---------+---------+
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |   58274 |
| 2014-05-12 12:48:13 | tricia  | mars    | gene    | venus   |  194925 |
| 2014-05-12 15:02:49 | phil    | mars    | phil    | saturn  |    1048 |
| 2014-05-12 18:59:18 | barb    | saturn  | tricia  | venus   |     271 |
```

```
| 2014-05-14 09:31:37 | gene    | venus   | barb    | mars    |    2291 |
| 2014-05-14 11:52:17 | phil    | mars    | tricia  | saturn  |    5781 |
| 2014-05-14 14:42:21 | barb    | venus   | barb    | venus   |   98151 |
| 2014-05-14 17:03:01 | tricia  | saturn  | phil    | venus   | 2394482 |
| 2014-05-15 07:17:48 | gene    | mars    | gene    | saturn  |    3824 |
| 2014-05-15 08:50:57 | phil    | venus   | phil    | venus   |     978 |
| 2014-05-15 10:25:52 | gene    | mars    | tricia  | saturn  |  998532 |
| 2014-05-15 17:35:31 | gene    | saturn  | gene    | mars    |    3856 |
| 2014-05-16 09:00:28 | gene    | venus   | barb    | mars    |     613 |
| 2014-05-16 23:04:19 | phil    | venus   | barb    | venus   |   10294 |
| 2014-05-19 12:49:23 | phil    | mars    | tricia  | saturn  |     873 |
| 2014-05-19 22:21:51 | gene    | saturn  | gene    | venus   |   23992 |
+---------------------+---------+---------+---------+---------+---------+
```

To create and load the `mail` table, change location into the *tables* directory of the `recipes` distribution and run this command:

```
% mysql cookbook < mail.sql
```

This chapter also uses other tables from time to time. Some were used in previous chapters, whereas others are new. To create any of them, do so the same way as for the `mail` table, using the appropriate script in the *tables* directory. In addition, many of the other scripts and programs used in this chapter are located in the *select* directory. The files in that directory enable you to try the examples more easily.

Many of the statements shown here can be executed from within the *mysql* program, which is discussed in Chapter 1. A few examples involve issuing statements from within the context of a programming language. See Chapter 2 for information on programming techniques.

3.1. Specifying Which Columns and Rows to Select

Problem

You want to display specific columns and rows from a table.

Solution

To indicate which columns to display, name them in the output column list. To indicate which rows to display, use a `WHERE` clause that specifies conditions that rows must satisfy.

Discussion

The simplest way to display columns from a table is to use `SELECT * FROM` *`tbl_name`*. The `*` specifier is a shortcut that means "all columns":

```
mysql> SELECT * FROM mail;
+---------------------+---------+---------+---------+---------+---------+
| t                   | srcuser | srchost | dstuser | dsthost | size    |
+---------------------+---------+---------+---------+---------+---------+
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |   58274 |
| 2014-05-12 12:48:13 | tricia  | mars    | gene    | venus   |  194925 |
| 2014-05-12 15:02:49 | phil    | mars    | phil    | saturn  |    1048 |
| 2014-05-12 18:59:18 | barb    | saturn  | tricia  | venus   |     271 |
…
```

Using `*` is easy, but you cannot select only certain columns or control column display order. Naming columns explicitly enables you to select only the ones of interest, in any order. This query omits the recipient columns and displays the sender before the date and size:

```
mysql> SELECT srcuser, srchost, t, size FROM mail;
+---------+---------+---------------------+---------+
| srcuser | srchost | t                   | size    |
+---------+---------+---------------------+---------+
| barb    | saturn  | 2014-05-11 10:15:08 |   58274 |
| tricia  | mars    | 2014-05-12 12:48:13 |  194925 |
| phil    | mars    | 2014-05-12 15:02:49 |    1048 |
| barb    | saturn  | 2014-05-12 18:59:18 |     271 |
…
```

Unless you qualify or restrict a `SELECT` query in some way, it retrieves every row in your table. To be more precise, provide a `WHERE` clause that specifies one or more conditions that rows must satisfy.

Conditions can test for equality, inequality, or relative ordering. For some types of data, such as strings, you can use pattern matches. The following statements select columns from rows in the `mail` table containing `srchost` values that are exactly equal to the string `'venus'` or that begin with the letter `'s'`:

```
mysql> SELECT t, srcuser, srchost  FROM mail WHERE srchost = 'venus';
+---------------------+---------+---------+
| t                   | srcuser | srchost |
+---------------------+---------+---------+
| 2014-05-14 09:31:37 | gene    | venus   |
| 2014-05-14 14:42:21 | barb    | venus   |
| 2014-05-15 08:50:57 | phil    | venus   |
| 2014-05-16 09:00:28 | gene    | venus   |
| 2014-05-16 23:04:19 | phil    | venus   |
+---------------------+---------+---------+
mysql> SELECT t, srcuser, srchost FROM mail WHERE srchost LIKE 's%';
+---------------------+---------+---------+
| t                   | srcuser | srchost |
+---------------------+---------+---------+
| 2014-05-11 10:15:08 | barb    | saturn  |
| 2014-05-12 18:59:18 | barb    | saturn  |
| 2014-05-14 17:03:01 | tricia  | saturn  |
| 2014-05-15 17:35:31 | gene    | saturn  |
```

```
| 2014-05-19 22:21:51 | gene    | saturn  |
+---------------------+---------+---------+
```

The `LIKE` operator in the previous query performs a pattern match, where `%` acts as a wildcard that matches any string. Recipe 5.8 discusses pattern matching further.

A `WHERE` clause can test multiple conditions and different conditions can test different columns. The following statement finds messages sent by `barb` to `tricia`:

```
mysql> SELECT * FROM mail WHERE srcuser = 'barb' AND dstuser = 'tricia';
+---------------------+---------+---------+---------+---------+-------+
| t                   | srcuser | srchost | dstuser | dsthost | size  |
+---------------------+---------+---------+---------+---------+-------+
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    | 58274 |
| 2014-05-12 18:59:18 | barb    | saturn  | tricia  | venus   |   271 |
+---------------------+---------+---------+---------+---------+-------+
```

Output columns can be calculated by evaluating expressions. This query combines the `srcuser` and `srchost` columns using `CONCAT()` to produce composite values in email address format:

```
mysql> SELECT t, CONCAT(srcuser,'@',srchost), size FROM mail;
+---------------------+----------------------------+---------+
| t                   | CONCAT(srcuser,'@',srchost) | size    |
+---------------------+----------------------------+---------+
| 2014-05-11 10:15:08 | barb@saturn                |   58274 |
| 2014-05-12 12:48:13 | tricia@mars                |  194925 |
| 2014-05-12 15:02:49 | phil@mars                  |    1048 |
| 2014-05-12 18:59:18 | barb@saturn                |     271 |
…
```

You'll notice that the email address column label is the expression that calculates it. To provide a better label, use a column alias (see Recipe 3.2).

3.2. Naming Query Result Columns

Problem

The column names in a query result are unsuitable, ugly, or difficult to work with.

Solution

Use aliases to choose your own column names.

Discussion

When you retrieve a result set, MySQL gives every output column a name. (That's how the *mysql* program gets the names you see displayed in the initial row of column headers in result set output.) By default, MySQL assigns the column names specified in the

CREATE TABLE or ALTER TABLE statement to output columns, but if these defaults are not suitable, you can use column aliases to specify your own names.

This section explains aliases and shows how to use them to assign column names in statements. If you're writing a program that must determine the names, see Recipe 10.2 for information about accessing column metadata.

If an output column comes directly from a table, MySQL uses the table column name for the output column name. The following statement selects four table columns, the names of which become the corresponding output column names:

```
mysql> SELECT t, srcuser, srchost, size FROM mail;
+---------------------+---------+---------+---------+
| t                   | srcuser | srchost | size    |
+---------------------+---------+---------+---------+
| 2014-05-11 10:15:08 | barb    | saturn  |   58274 |
| 2014-05-12 12:48:13 | tricia  | mars    |  194925 |
| 2014-05-12 15:02:49 | phil    | mars    |    1048 |
| 2014-05-12 18:59:18 | barb    | saturn  |     271 |
…
```

If you generate a column by evaluating an expression, the expression itself is the column name. This can produce long and unwieldy names in result sets, as illustrated by the following statement that uses one expression to reformat the dates in the `t` column, and another to combine `srcuser` and `srchost` into email address format:

```
mysql> SELECT
    -> DATE_FORMAT(t,'%M %e, %Y'), CONCAT(srcuser,'@',srchost), size
    -> FROM mail;
+----------------------------+----------------------------+---------+
| DATE_FORMAT(t,'%M %e, %Y') | CONCAT(srcuser,'@',srchost) | size    |
+----------------------------+----------------------------+---------+
| May 11, 2014               | barb@saturn                |   58274 |
| May 12, 2014               | tricia@mars                |  194925 |
| May 12, 2014               | phil@mars                  |    1048 |
| May 12, 2014               | barb@saturn                |     271 |
…
```

To choose your own output column name, use an AS *name* clause to specify a column alias (the keyword AS is optional). The following statement retrieves the same result as the previous one, but renames the first column to `date_sent` and the second to `sender`:

```
mysql> SELECT
    -> DATE_FORMAT(t,'%M %e, %Y') AS date_sent,
    -> CONCAT(srcuser,'@',srchost) AS sender,
    -> size FROM mail;
+--------------+--------------+---------+
| date_sent    | sender       | size    |
+--------------+--------------+---------+
| May 11, 2014 | barb@saturn  |   58274 |
| May 12, 2014 | tricia@mars  |  194925 |
| May 12, 2014 | phil@mars    |    1048 |
```

```
| May 12, 2014 | barb@saturn   |     271 |
…
```

The aliases make the column names more concise, easier to read, and more meaningful. Aliases are subject to a few restrictions. For example, they must be quoted if they are SQL keywords, entirely numeric, or contain spaces or other special characters (an alias can consist of several words if you want to use a descriptive phrase). The following statement retrieves the same data values as the preceding one but uses phrases to name the output columns:

```
mysql> SELECT
    -> DATE_FORMAT(t,'%M %e, %Y') AS 'Date of message',
    -> CONCAT(srcuser,'@',srchost) AS 'Message sender',
    -> size AS 'Number of bytes' FROM mail;
+-----------------+----------------+-----------------+
| Date of message | Message sender | Number of bytes |
+-----------------+----------------+-----------------+
| May 11, 2014    | barb@saturn    |           58274 |
| May 12, 2014    | tricia@mars    |          194925 |
| May 12, 2014    | phil@mars      |            1048 |
| May 12, 2014    | barb@saturn    |             271 |
…
```

If MySQL complains about a single-word alias, the word probably is reserved. Quoting the alias should make it legal:

```
mysql> SELECT 1 AS INTEGER;
You have an error in your SQL syntax near 'INTEGER'
mysql> SELECT 1 AS 'INTEGER';
+---------+
| INTEGER |
+---------+
|       1 |
+---------+
```

Column aliases also are useful for programming purposes. If you write a program that fetches rows into an array and accesses them by numeric column indexes, the presence or absence of column aliases makes no difference because aliases don't change the positions of columns within the result set. However, aliases make a big difference if you access output columns by name because aliases change those names. Exploit this fact to give your program easier names to work with. For example, if your query displays reformatted message time values from the `mail` table using the expression `DATE_FORMAT(t,'%M %e, %Y')`, that expression is also the name you must use when referring to the output column. In a Perl hashref, for example, you'd access it as `$ref->{"DATE_FORMAT(t,'%M %e, %Y')"}`. That's inconvenient. Use `AS date_sent` to give the column an alias and you can refer to it more easily as `$ref->{date_sent}`. Here's an example that shows how a Perl DBI script might process such values. It retrieves rows into a hash and refers to column values by name:

```
$sth = $dbh->prepare ("SELECT srcuser,
                       DATE_FORMAT(t,'%M %e, %Y') AS date_sent
                       FROM mail");
$sth->execute ();
while (my $ref = $sth->fetchrow_hashref ())
{
  printf "user: %s, date sent: %s\n", $ref->{srcuser}, $ref->{date_sent};
}
```

In Java, you'd do something like this, where the argument to `getString()` names the column to access:

```
Statement s = conn.createStatement ();
s.executeQuery ("SELECT srcuser,"
                + " DATE_FORMAT(t,'%M %e, %Y') AS date_sent"
                + " FROM mail");
ResultSet rs = s.getResultSet ();
while (rs.next ())  // loop through rows of result set
{
  String name = rs.getString ("srcuser");
  String dateSent = rs.getString ("date_sent");
  System.out.println ("user: " + name + ", date sent: " + dateSent);
}
rs.close ();
s.close ();
```

Recipe 2.4 shows for each of our programming languages how to fetch rows into data structures that permit access to column values by name. The *select* directory of the `recipes` distribution has examples that show how to do this for the `mail` table.

You cannot refer to column aliases in a `WHERE` clause. Thus, the following statement is illegal:

```
mysql> SELECT t, srcuser, dstuser, size/1024 AS kilobytes
    -> FROM mail WHERE kilobytes > 500;
ERROR 1054 (42S22): Unknown column 'kilobytes' in 'where clause'
```

The error occurs because an alias names an *output* column, whereas a `WHERE` clause operates on *input* columns to determine which rows to select for output. To make the statement legal, replace the alias in the `WHERE` clause with the same column or expression that the alias represents:

```
mysql> SELECT t, srcuser, dstuser, size/1024 AS kilobytes
    -> FROM mail WHERE size/1024 > 500;
+---------------------+---------+---------+-----------+
| t                   | srcuser | dstuser | kilobytes |
+---------------------+---------+---------+-----------+
| 2014-05-14 17:03:01 | tricia  | phil    | 2338.3613 |
| 2014-05-15 10:25:52 | gene    | tricia  |  975.1289 |
+---------------------+---------+---------+-----------+
```

3.3. Sorting Query Results

Problem

Your query results aren't sorted the way you want.

Solution

MySQL can't read your mind. Use an `ORDER BY` clause to tell it how to sort result rows.

Discussion

When you select rows, the MySQL server is free to return them in any order unless you instruct it otherwise by saying how to sort the result. There are lots of ways to use sorting techniques, as Chapter 7 explores in detail. Briefly, to sort a result set, add an `ORDER BY` clause that names the column or columns to use for sorting. This statement names multiple columns in the `ORDER BY` clause to sort rows by host and by user within each host:

```
mysql> SELECT * FROM mail WHERE dstuser = 'tricia'
    -> ORDER BY srchost, srcuser;
+---------------------+---------+---------+---------+---------+--------+
| t                   | srcuser | srchost | dstuser | dsthost | size   |
+---------------------+---------+---------+---------+---------+--------+
| 2014-05-15 10:25:52 | gene    | mars    | tricia  | saturn  | 998532 |
| 2014-05-14 11:52:17 | phil    | mars    | tricia  | saturn  |   5781 |
| 2014-05-19 12:49:23 | phil    | mars    | tricia  | saturn  |    873 |
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |  58274 |
| 2014-05-12 18:59:18 | barb    | saturn  | tricia  | venus   |    271 |
+---------------------+---------+---------+---------+---------+--------+
```

To sort a column in reverse (descending) order, add the keyword `DESC` after its name in the `ORDER BY` clause:

```
mysql> SELECT * FROM mail WHERE size > 50000 ORDER BY size DESC;
+---------------------+---------+---------+---------+---------+---------+
| t                   | srcuser | srchost | dstuser | dsthost | size    |
+---------------------+---------+---------+---------+---------+---------+
| 2014-05-14 17:03:01 | tricia  | saturn  | phil    | venus   | 2394482 |
| 2014-05-15 10:25:52 | gene    | mars    | tricia  | saturn  |  998532 |
| 2014-05-12 12:48:13 | tricia  | mars    | gene    | venus   |  194925 |
| 2014-05-14 14:42:21 | barb    | venus   | barb    | venus   |   98151 |
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |   58274 |
+---------------------+---------+---------+---------+---------+---------+
```

3.4. Removing Duplicate Rows

Problem

Output from a query contains duplicate rows. You want to eliminate them.

Solution

Use `DISTINCT`.

Discussion

Some queries produce results containing duplicate rows. For example, to see who sent mail, query the `mail` table like this:

```
mysql> SELECT srcuser FROM mail;
+---------+
| srcuser |
+---------+
| barb    |
| tricia  |
| phil    |
| barb    |
| gene    |
| phil    |
| barb    |
| tricia  |
| gene    |
| phil    |
| gene    |
| gene    |
| gene    |
| phil    |
| phil    |
| gene    |
+---------+
```

That result is heavily redundant. To remove the duplicate rows and produce a set of unique values, add `DISTINCT` to the query:

```
mysql> SELECT DISTINCT srcuser FROM mail;
+---------+
| srcuser |
+---------+
| barb    |
| tricia  |
| phil    |
| gene    |
+---------+
```

To count the number of unique values in a column, use `COUNT(DISTINCT)`:

```
mysql> SELECT COUNT(DISTINCT srcuser) FROM mail;
+-------------------------+
| COUNT(DISTINCT srcuser) |
+-------------------------+
|                       4 |
+-------------------------+
```

`DISTINCT` works with multiple-column output, too. The following query shows which dates are represented in the `mail` table:

```
mysql> SELECT DISTINCT YEAR(t), MONTH(t), DAYOFMONTH(t) FROM mail;
+---------+----------+---------------+
| YEAR(t) | MONTH(t) | DAYOFMONTH(t) |
+---------+----------+---------------+
|    2014 |        5 |            11 |
|    2014 |        5 |            12 |
|    2014 |        5 |            14 |
|    2014 |        5 |            15 |
|    2014 |        5 |            16 |
|    2014 |        5 |            19 |
+---------+----------+---------------+
```

See Also

Chapter 8 revisits `DISTINCT` and `COUNT(DISTINCT)`. Chapter 16 discusses duplicate removal in more detail.

3.5. Working with NULL Values

Problem

You're trying to to compare column values to `NULL`, but it isn't working.

Solution

Use the proper comparison operators: `IS NULL`, `IS NOT NULL`, or `<=>`.

Discussion

Conditions that involve `NULL` are special because `NULL` means "unknown value." Consequently, comparisons such as *`value`* `= NULL` or *`value`* `<> NULL` always produce a result of `NULL` (not true or false) because it's impossible to tell whether they are true or false. Even `NULL = NULL` produces `NULL` because you can't determine whether one unknown value is the same as another.

To look for values that are or are not `NULL`, use the `IS NULL` or `IS NOT NULL` operator. Suppose that a table named `expt` contains experimental results for subjects who are to be given four tests each and that represents tests not yet administered using `NULL`:

```
+---------+------+-------+
| subject | test | score |
+---------+------+-------+
| Jane    | A    |    47 |
| Jane    | B    |    50 |
| Jane    | C    |  NULL |
| Jane    | D    |  NULL |
| Marvin  | A    |    52 |
| Marvin  | B    |    45 |
| Marvin  | C    |    53 |
| Marvin  | D    |  NULL |
+---------+------+-------+
```

You can see that = and <> fail to identify NULL values:

```
mysql> SELECT * FROM expt WHERE score = NULL;
Empty set (0.00 sec)
mysql> SELECT * FROM expt WHERE score <> NULL;
Empty set (0.00 sec)
```

Write the statements like this instead:

```
mysql> SELECT * FROM expt WHERE score IS NULL;
+---------+------+-------+
| subject | test | score |
+---------+------+-------+
| Jane    | C    |  NULL |
| Jane    | D    |  NULL |
| Marvin  | D    |  NULL |
+---------+------+-------+
mysql> SELECT * FROM expt WHERE score IS NOT NULL;
+---------+------+-------+
| subject | test | score |
+---------+------+-------+
| Jane    | A    |    47 |
| Jane    | B    |    50 |
| Marvin  | A    |    52 |
| Marvin  | B    |    45 |
| Marvin  | C    |    53 |
+---------+------+-------+
```

The MySQL-specific <=> comparison operator, unlike the = operator, is true even for two NULL values:

```
mysql> SELECT NULL = NULL, NULL <=> NULL;
+-------------+---------------+
| NULL = NULL | NULL <=> NULL |
+-------------+---------------+
|        NULL |             1 |
+-------------+---------------+
```

Sometimes it's useful to map NULL values onto some other value that has more meaning in the context of your application. For example, use IF() to map NULL onto the string Unknown:

```
mysql> SELECT subject, test, IF(score IS NULL,'Unknown', score) AS 'score'
    -> FROM expt;
+---------+------+---------+
| subject | test | score   |
+---------+------+---------+
| Jane    | A    | 47      |
| Jane    | B    | 50      |
| Jane    | C    | Unknown |
| Jane    | D    | Unknown |
| Marvin  | A    | 52      |
| Marvin  | B    | 45      |
| Marvin  | C    | 53      |
| Marvin  | D    | Unknown |
+---------+------+---------+
```

This `IF()`-based mapping technique works for any kind of value, but it's especially useful with `NULL` values because `NULL` tends to be given a variety of meanings: unknown, missing, not yet determined, out of range, and so forth. Choose the label that makes the most sense in a given context.

The preceding query can be written more concisely using `IFNULL()`, which tests its first argument and returns it if it's not `NULL`, or returns its second argument otherwise:

```
SELECT subject, test, IFNULL(score,'Unknown') AS 'score'
FROM expt;
```

In other words, these two tests are equivalent:

```
IF(expr1 IS NOT NULL,expr1,expr2)
IFNULL(expr1,expr2)
```

From a readability standpoint, `IF()` often is easier to understand than `IFNULL()`. From a computational perspective, `IFNULL()` is more efficient because *`expr1`* need not be evaluated twice, as happens with `IF()`.

See Also

`NULL` values also behave specially with respect to sorting and summary operations. See Recipes 7.11 and 8.6.

3.6. Writing Comparisons Involving NULL in Programs

Problem

You're writing a program that looks for rows containing a specific value, but it fails when the value is `NULL`.

Solution

Choose the proper comparison operator according to whether the comparison value is or is not NULL.

Discussion

Recipe 3.5 discusses the need to use different comparison operators for NULL values than for non-NULL values in SQL statements. This issue leads to a subtle danger when constructing statement strings within programs. If a value stored in a variable might represent a NULL value, you must account for that when you use the value in comparisons. For example, in Perl, undef represents a NULL value, so to construct a statement that finds rows in the expt table matching some arbitrary value in a $score variable, you cannot do this:

```
$sth = $dbh->prepare ("SELECT * FROM expt WHERE score = ?");
$sth->execute ($score);
```

The statement fails when $score is undef because the resulting statement becomes:

```
SELECT * FROM expt WHERE score = NULL
```

A comparison of score = NULL is never true, so that statement returns no rows. To take into account the possibility that $score could be undef, construct the statement using the appropriate comparison operator like this:

```
$operator = defined ($score) ? "=" : "IS";
$sth = $dbh->prepare ("SELECT * FROM expt WHERE score $operator ?");
$sth->execute ($score);
```

This results in statements as follows for $score values of undef (NULL) or 43 (not NULL):

```
SELECT * FROM expt WHERE score IS NULL
SELECT * FROM expt WHERE score = 43
```

For inequality tests, set $operator like this instead:

```
$operator = defined ($score) ? "<>" : "IS NOT";
```

3.7. Using Views to Simplify Table Access

Problem

You want to refer to values calculated from expressions without writing the expressions each time you retrieve them.

Solution

Use a view defined such that its columns perform the desired calculations.

Discussion

Suppose that you retrieve several values from the `mail` table, using expressions to calculate most of them:

```
mysql> SELECT
    -> DATE_FORMAT(t,'%M %e, %Y') AS date_sent,
    -> CONCAT(srcuser,'@',srchost) AS sender,
    -> CONCAT(dstuser,'@',dsthost) AS recipient,
    -> size FROM mail;
+--------------+---------------+---------------+---------+
| date_sent    | sender        | recipient     | size    |
+--------------+---------------+---------------+---------+
| May 11, 2014 | barb@saturn   | tricia@mars   |   58274 |
| May 12, 2014 | tricia@mars   | gene@venus    |  194925 |
| May 12, 2014 | phil@mars     | phil@saturn   |    1048 |
| May 12, 2014 | barb@saturn   | tricia@venus  |     271 |
…
```

If you must issue such a statement often, it's inconvenient to keep writing the expressions. To make the statement results easier to access, use a view, which is a virtual table that contains no data. Instead, it's defined as the `SELECT` statement that retrieves the data of interest. The following view, `mail_view`, is equivalent to the `SELECT` statement just shown:

```
mysql> CREATE VIEW mail_view AS
    -> SELECT
    -> DATE_FORMAT(t,'%M %e, %Y') AS date_sent,
    -> CONCAT(srcuser,'@',srchost) AS sender,
    -> CONCAT(dstuser,'@',dsthost) AS recipient,
    -> size FROM mail;
```

To access the view contents, refer to it like any other table. You can select some or all of its columns, add a `WHERE` clause to restrict which rows to retrieve, use `ORDER BY` to sort the rows, and so forth. For example:

```
mysql> SELECT date_sent, sender, size FROM mail_view
    -> WHERE size > 100000 ORDER BY size;
+--------------+---------------+---------+
| date_sent    | sender        | size    |
+--------------+---------------+---------+
| May 12, 2014 | tricia@mars   |  194925 |
| May 15, 2014 | gene@mars     |  998532 |
| May 14, 2014 | tricia@saturn | 2394482 |
+--------------+---------------+---------+
```

Stored programs provide another way to encapsulate calculations (see Recipe 9.2).

3.8. Selecting Data from Multiple Tables

Problem

The answer to a question requires data from more than one table.

Solution

Use a join or a subquery.

Discussion

The queries shown so far select data from a single table, but sometimes you must retrieve information from multiple tables. Two types of statements that accomplish this are joins and subqueries. A join matches rows in one table with rows in another and enables you to retrieve output rows that contain columns from either or both tables. A subquery is one query nested within another, to perform a comparison between values selected by the inner query against values selected by the outer query.

This recipe shows a couple brief examples to illustrate the basic ideas. Other examples appear elsewhere: subqueries are used in various examples throughout the book (for example, Recipes 3.10 and 8.3). Chapter 14 discusses joins in detail, including some that select from more than two tables.

The following examples use the `profile` table introduced in Chapter 2. Recall that it lists the people on your buddy list:

```
mysql> SELECT * FROM profile;
+----+---------+------------+-------+----------------------+------+
| id | name    | birth      | color | foods                | cats |
+----+---------+------------+-------+----------------------+------+
|  1 | Sybil   | 1970-04-13 | black | lutefisk,fadge,pizza |    0 |
|  2 | Nancy   | 1969-09-30 | white | burrito,curry,eggroll |    3 |
|  3 | Ralph   | 1973-11-02 | red   | eggroll,pizza        |    4 |
|  4 | Lothair | 1963-07-04 | blue  | burrito,curry        |    5 |
|  5 | Henry   | 1965-02-14 | red   | curry,fadge          |    1 |
|  6 | Aaron   | 1968-09-17 | green | lutefisk,fadge       |    1 |
|  7 | Joanna  | 1952-08-20 | green | lutefisk,fadge       |    0 |
|  8 | Stephen | 1960-05-01 | white | burrito,pizza        |    0 |
+----+---------+------------+-------+----------------------+------+
```

Let's extend use of the `profile` table to include another table named `profile_contact`. This second table indicates how to contact people listed in the `profile` table via various social media services and is defined like this:

```
CREATE TABLE profile_contact
(
  profile_id   INT UNSIGNED NOT NULL, # ID from profile table
  service      VARCHAR(20) NOT NULL,  # social media service name
```

```
  contact_name VARCHAR(25) NOT NULL,  # name to use for contacting person
  INDEX (profile_id)
);
```

The table associates each row with the proper `profile` row via the `profile_id` column. The `service` and `contact_name` columns name the media service and the name to use for contacting the given person via that service. For the examples, assume that the table contains these rows:

```
mysql> SELECT * FROM profile_contact ORDER BY profile_id, service;
+------------+----------+--------------+
| profile_id | service  | contact_name |
+------------+----------+--------------+
|          1 | Facebook | user1-fbid   |
|          1 | Twitter  | user1-twtrid |
|          2 | Facebook | user2-msnid  |
|          2 | LinkedIn | user2-lnkdid |
|          2 | Twitter  | user2-fbrid  |
|          4 | LinkedIn | user4-lnkdid |
+------------+----------+--------------+
```

A question that requires information from both tables is, "For each person in the `profile` table, show me which services I can use to get in touch, and the contact name for each service." To answer this question, use a join. Select from both tables and match rows by comparing the `id` column from the `profile` table with the `profile_id` column from the `profile_contact` table:

```
mysql> SELECT id, name, service, contact_name
    -> FROM profile INNER JOIN profile_contact ON id = profile_id;
+----+---------+----------+--------------+
| id | name    | service  | contact_name |
+----+---------+----------+--------------+
|  1 | Sybil   | Twitter  | user1-twtrid |
|  1 | Sybil   | Facebook | user1-fbid   |
|  2 | Nancy   | Twitter  | user2-fbrid  |
|  2 | Nancy   | Facebook | user2-msnid  |
|  2 | Nancy   | LinkedIn | user2-lnkdid |
|  4 | Lothair | LinkedIn | user4-lnkdid |
+----+---------+----------+--------------+
```

The `FROM` clause indicates the tables from which to select data, and the `ON` clause tells MySQL which columns to use to find matches between the tables. In the result, rows include the `id` and `name` columns from the `profile` table, and the `service` and `contact_name` columns from the `profile_contact` table.

Here's another question that requires both tables to answer: "List all the `profile_contact` records for Nancy." To pull the proper rows from the `profile_contact` table, you need Nancy's ID, which is stored in the `profile` table. To write the query without looking up Nancy's ID yourself, use a subquery that, given her name, looks it up for you:

```
mysql> SELECT * FROM profile_contact
    -> WHERE profile_id = (SELECT id FROM profile WHERE name = 'Nancy');
+------------+----------+--------------+
| profile_id | service  | contact_name |
+------------+----------+--------------+
|          2 | Twitter  | user2-fbrid  |
|          2 | Facebook | user2-msnid  |
|          2 | LinkedIn | user2-lnkdid |
+------------+----------+--------------+
```

Here the subquery appears as a nested `SELECT` statement enclosed within parentheses.

3.9. Selecting Rows from the Beginning, End, or Middle of Query Results

Problem

You want only certain rows from a result set, such as the first one, the last five, or rows 21 through 40.

Solution

Use a `LIMIT` clause, perhaps in conjunction with an `ORDER BY` clause.

Discussion

MySQL supports a `LIMIT` clause that tells the server to return only part of a result set. `LIMIT` is a MySQL-specific extension to SQL that is extremely valuable when your result set contains more rows than you want to see at a time. It enables you to retrieve an arbitrary section of a result set. Typical `LIMIT` uses include the following kinds of problems:

- Answering questions about first or last, largest or smallest, newest or oldest, least or most expensive, and so forth.
- Splitting a result set into sections so that you can process it one piece at a time. This technique is common in web applications for displaying a large search result across several pages. Showing the result in sections enables display of smaller, easier-to-understand pages.

The following examples use the `profile` table shown in Recipe 3.8. To see the first *n* rows of a `SELECT` result, add `LIMIT` *n* to the end of the statement:

```
mysql> SELECT * FROM profile LIMIT 1;
+----+-------+------------+-------+---------------------+------+
| id | name  | birth      | color | foods               | cats |
+----+-------+------------+-------+---------------------+------+
```

```
|  1 | Sybil | 1970-04-13 | black | lutefisk,fadge,pizza |    0 |
+----+-------+------------+-------+----------------------+------+
mysql> SELECT * FROM profile LIMIT 3;
+----+-------+------------+-------+----------------------+------+
| id | name  | birth      | color | foods                | cats |
+----+-------+------------+-------+----------------------+------+
|  1 | Sybil | 1970-04-13 | black | lutefisk,fadge,pizza |    0 |
|  2 | Nancy | 1969-09-30 | white | burrito,curry,eggroll |    3 |
|  3 | Ralph | 1973-11-02 | red   | eggroll,pizza        |    4 |
+----+-------+------------+-------+----------------------+------+
```

`LIMIT` *n* means "return *at most n* rows." If you specify `LIMIT 10`, and the result set has only four rows, the server returns four rows.

The rows in the preceding query results are returned in no particular order, so they may not be very meaningful. A more common technique uses `ORDER BY` to sort the result set and `LIMIT` to find smallest and largest values. For example, to find the row with the minimum (earliest) birth date, sort by the `birth` column, then add `LIMIT 1` to retrieve the first row:

```
mysql> SELECT * FROM profile ORDER BY birth LIMIT 1;
+----+--------+------------+-------+---------------+------+
| id | name   | birth      | color | foods         | cats |
+----+--------+------------+-------+---------------+------+
|  7 | Joanna | 1952-08-20 | green | lutefisk,fadge |    0 |
+----+--------+------------+-------+---------------+------+
```

This works because MySQL processes the `ORDER BY` clause to sort the rows, then applies `LIMIT`.

To obtain rows from the end of a result set, sort them in the opposite order. The statement that finds the row with the most recent birth date is similar to the previous one, except that the sort order is descending:

```
mysql> SELECT * FROM profile ORDER BY birth DESC LIMIT 1;
+----+-------+------------+-------+--------------+------+
| id | name  | birth      | color | foods        | cats |
+----+-------+------------+-------+--------------+------+
|  3 | Ralph | 1973-11-02 | red   | eggroll,pizza |    4 |
+----+-------+------------+-------+--------------+------+
```

To find the earliest or latest birthday within the calendar year, sort by the month and day of the `birth` values:

```
mysql> SELECT name, DATE_FORMAT(birth,'%m-%d') AS birthday
    -> FROM profile ORDER BY birthday LIMIT 1;
+-------+----------+
| name  | birthday |
+-------+----------+
| Henry | 02-14    |
+-------+----------+
```

You can obtain the same information by running these statements without `LIMIT` and ignoring everything but the first row. The advantage of `LIMIT` is that the server returns only the first row, and the extra rows don't cross the network at all. This is much more efficient than retrieving an entire result set, only to discard all but one row.

To pull rows from the middle of a result set, use the two-argument form of `LIMIT`, which enables you to pick an arbitrary section of rows. The arguments indicate how many rows to skip and how many to return. This means that you can use `LIMIT` to do such things as skip two rows and return the next one, thus answering questions such as "What is the *third*-smallest or *third*-largest value?" These are questions that `MIN()` or `MAX()` are not suited for, but are easy with `LIMIT`:

```
mysql> SELECT * FROM profile ORDER BY birth LIMIT 2,1;
+----+---------+------------+-------+--------------+------+
| id | name    | birth      | color | foods        | cats |
+----+---------+------------+-------+--------------+------+
|  4 | Lothair | 1963-07-04 | blue  | burrito,curry |    5 |
+----+---------+------------+-------+--------------+------+
mysql> SELECT * FROM profile ORDER BY birth DESC LIMIT 2,1;
+----+-------+------------+-------+----------------------+------+
| id | name  | birth      | color | foods                | cats |
+----+-------+------------+-------+----------------------+------+
|  2 | Nancy | 1969-09-30 | white | burrito,curry,eggroll |    3 |
+----+-------+------------+-------+----------------------+------+
```

The two-argument form of `LIMIT` also makes it possible to partition a result set into smaller sections. For example, to retrieve 20 rows at a time from a result, issue a `SELECT` statement repeatedly, but vary its `LIMIT` clause like so:

```
SELECT ... FROM ... ORDER BY ... LIMIT 0, 20;
SELECT ... FROM ... ORDER BY ... LIMIT 20, 20;
SELECT ... FROM ... ORDER BY ... LIMIT 40, 20;
…
```

Web developers often use `LIMIT` this way to split a large search result into smaller, more manageable pieces so that it can be presented over several pages. Recipe 20.10 discusses this technique further.

To determine the number of rows in a result set so that you can determine the number of sections, issue a `COUNT()` statement first. For example, to display `profile` table rows in name order, three at a time, you can find out how many there are with the following statement:

```
mysql> SELECT COUNT(*) FROM profile;
+----------+
| COUNT(*) |
+----------+
|        8 |
+----------+
```

That tells you that there are three sets of rows (the last with fewer than three rows), which you can retrieve as follows:

```
SELECT * FROM profile ORDER BY name LIMIT 0, 3;
SELECT * FROM profile ORDER BY name LIMIT 3, 3;
SELECT * FROM profile ORDER BY name LIMIT 6, 3;
```

You can also fetch the first part of a result set and determine at the same time how big the result would have been without the `LIMIT` clause. To fetch the first three rows from the `profile` table, and then obtain the size of the full result, run these statements:

```
SELECT SQL_CALC_FOUND_ROWS * FROM profile ORDER BY name LIMIT 4;
SELECT FOUND_ROWS();
```

The keyword `SQL_CALC_FOUND_ROWS` in the first statement tells MySQL to calculate the size of the entire result set even though the statement requests that only part of it be returned. The row count is available by calling `FOUND_ROWS()`. If that function returns a value greater than three, there are other rows yet to be retrieved.

See Also

`LIMIT` is useful in combination with `RAND()` to make random selections from a set of items. See Recipe 15.8.

You can use `LIMIT` to restrict the effect of a `DELETE` or `UPDATE` statement to a subset of the rows that would otherwise be deleted or updated, respectively. For more information about using `LIMIT` for duplicate row removal, see Recipe 16.4.

3.10. What to Do When LIMIT Requires the "Wrong" Sort Order

Problem

`LIMIT` usually works best in conjunction with an `ORDER BY` clause that sorts rows. But sometimes that sort order differs from what you want for the final result.

Solution

Use `LIMIT` in a subquery to retrieve the desired rows, then use the outer query to sort them.

Discussion

If you want the last four rows of a result set, you can obtain them easily by sorting the set in reverse order and using `LIMIT 4`. The following statement returns the names and birth dates for the four people in the `profile` table who were born most recently:

```
mysql> SELECT name, birth FROM profile ORDER BY birth DESC LIMIT 4;
+-------+------------+
| name  | birth      |
+-------+------------+
| Ralph | 1973-11-02 |
| Sybil | 1970-04-13 |
| Nancy | 1969-09-30 |
| Aaron | 1968-09-17 |
+-------+------------+
```

But that requires sorting the `birth` values in descending order to place them at the head of the result set. What if you want the output rows to appear in ascending order instead? Use the `SELECT` as a subquery of an outer statement that re-sorts the rows in the desired final order:

```
mysql> SELECT * FROM
    -> (SELECT name, birth FROM profile ORDER BY birth DESC LIMIT 4) AS t
    -> ORDER BY birth;
+-------+------------+
| name  | birth      |
+-------+------------+
| Aaron | 1968-09-17 |
| Nancy | 1969-09-30 |
| Sybil | 1970-04-13 |
| Ralph | 1973-11-02 |
+-------+------------+
```

`AS t` is used here because any table referred to in the `FROM` clause must have a name, even a "derived" table produced from a subquery.

3.11. Calculating LIMIT Values from Expressions

Problem

You want to use expressions to specify the arguments for `LIMIT`.

Solution

Sadly, you cannot. `LIMIT` arguments must be literal integers—unless you issue the statement in a context that permits the statement string to be constructed dynamically. In that case, you can evaluate the expressions yourself and insert the resulting values into the statement string.

Discussion

Arguments to `LIMIT` must be literal integers, not expressions. Statements such as the following are illegal:

```
SELECT * FROM profile LIMIT 5+5;
SELECT * FROM profile LIMIT @skip_count, @show_count;
```

The same "no expressions permitted" principle applies if you use an expression to calculate a `LIMIT` value in a program that constructs a statement string. You must evaluate the expression first, and then place the resulting value in the statement. For example, if you produce a statement string in Perl or PHP as follows, an error will result when you attempt to execute the statement:

```
$str = "SELECT * FROM profile LIMIT $x + $y";
```

To avoid the problem, evaluate the expression first:

```
$z = $x + $y;
$str = "SELECT * FROM profile LIMIT $z";
```

Or do this (don't omit the parentheses or the expression won't evaluate properly):

```
$str = "SELECT * FROM profile LIMIT " . ($x + $y);
```

To construct a two-argument `LIMIT` clause, evaluate both expressions before placing them into the statement string.

Another issue related to `LIMIT` (or other syntax constructions that require literal integer values) occurs when you use prepared statements from an API that quotes all data values as strings when binding them to parameter markers. Suppose that you prepare and execute a statement like this in PDO:

```
$sth = $dbh->prepare ("SELECT * FROM profile LIMIT ?,?");
$sth->execute (array (2, 4));
```

To resulting statement is as follows, with quoted `LIMIT` arguments, so statement execution fails:

```
SELECT * FROM profile LIMIT '2','4'
```

To avoid this problem, evaluate the `LIMIT` arguments and place them in the statement yourself, as just described. Alternatively, if your API has type-hinting capability, use it to indicate that the `LIMIT` arguments are integers to prevent them from being quoted.

CHAPTER 4

Table Management

4.0. Introduction

This chapter covers topics that relate to creating and populating tables:

- Cloning a table
- Copying from one table to another
- Using temporary tables
- Generating unique table names
- Determining what storage engine a table uses or converting it from one storage engine to another

Many of the examples in this chapter use a table named `mail` containing rows that track mail message traffic between users on a set of hosts (see Recipe 3.0). To create and load this table, change location into the *tables* directory of the `recipes` distribution and run this command:

```
% mysql cookbook < mail.sql
```

4.1. Cloning a Table

Problem

You want to create a table that has exactly the same structure as an existing table.

Solution

Use `CREATE TABLE ... LIKE` to clone the table structure. To also copy some or all of the rows from the original table to the new one, use `INSERT INTO ... SELECT`.

Discussion

To create a new table that is just like an existing table, use this statement:

```
CREATE TABLE new_table LIKE original_table;
```

The structure of the new table is the same as that of the original table, with a few exceptions: `CREATE TABLE ... LIKE` does not copy foreign key definitions, and it doesn't copy any `DATA DIRECTORY` or `INDEX DIRECTORY` table options that the table might use.

The new table is empty. If you also want the contents to be the same as the original table, copy the rows using an `INSERT INTO ... SELECT` statement:

```
INSERT INTO new_table SELECT * FROM original_table;
```

To copy only part of the table, add an appropriate `WHERE` clause that identifies which rows to copy. For example, these statements create a copy of the `mail` table named `mail2`, populated only with the rows for mail sent by `barb`:

```
CREATE TABLE mail2 LIKE mail;
INSERT INTO mail2 SELECT * FROM mail WHERE srcuser = 'barb';
```

For more information about `INSERT ... SELECT`, see Recipe 4.2.

4.2. Saving a Query Result in a Table

Problem

You want to save the result from a `SELECT` statement to a table rather than display it.

Solution

If the table exists, retrieve rows into it using `INSERT INTO ... SELECT`. If the table does not exist, create it on the fly using `CREATE TABLE ... SELECT`.

Discussion

The MySQL server normally returns the result of a `SELECT` statement to the client that executed the statement. For example, when you execute a statement from within the *mysql* program, the server returns the result to *mysql*, which in turn displays it on the screen. It's possible to save the results of a `SELECT` statement in a table instead, which is useful in several ways:

- You can easily create a complete or partial copy of a table. If you're developing an algorithm that modifies a table, it's safer to work with a copy of a table so that you need not worry about the consequences of mistakes. If the original table is large, creating a partial copy can speed the development process because queries run against it take less time.

- For a data-loading operation based on information that might be malformed, load new rows into a temporary table, perform some preliminary checks, and correct the rows as necessary. When you're satisfied that the new rows are okay, copy them from the temporary table to your main table.
- Some applications maintain a large repository table and a smaller working table into which rows are inserted on a regular basis, copying the working table rows to the repository periodically and clearing the working table.
- To perform summary operations on a large table more efficiently, avoid running expensive summary operations repeatedly on it. Instead, select summary information once into a second table and use that for further analysis.

This section shows how to retrieve a result set into a table. The table names `src_tbl` and `dst_tbl` in the examples refer to the source table from which rows are selected and the destination table into which they are stored, respectively.

If the destination table already exists, use `INSERT ... SELECT` to copy the result set into it. For example, if `dst_tbl` contains an integer column `i` and a string column `s`, the following statement copies rows from `src_tbl` into `dst_tbl`, assigning column `val` to `i` and column `name` to `s`:

```
INSERT INTO dst_tbl (i, s) SELECT val, name FROM src_tbl;
```

The number of columns to be inserted must match the number of selected columns, with the correspondence between columns based on position rather than name. To copy all columns, you can shorten the statement to this form:

```
INSERT INTO dst_tbl SELECT * FROM src_tbl;
```

To copy only certain rows, add a `WHERE` clause that selects those rows:

```
INSERT INTO dst_tbl SELECT * FROM src_tbl
WHERE val > 100 AND name LIKE 'A%';
```

The `SELECT` statement can produce values from expressions, too. For example, the following statement counts the number of times each name occurs in `src_tbl` and stores both the counts and the names in `dst_tbl`:

```
INSERT INTO dst_tbl (i, s) SELECT COUNT(*), name
FROM src_tbl GROUP BY name;
```

If the destination table does not exist, create it first with a `CREATE TABLE` statement, then copy rows into it with `INSERT ... SELECT`. Alternatively, use `CREATE TABLE ... SELECT` to create the destination table directly from the result of the `SELECT`. For example, to create `dst_tbl` and copy the entire contents of `src_tbl` into it, do this:

```
CREATE TABLE dst_tbl SELECT * FROM src_tbl;
```

MySQL creates the columns in `dst_tbl` based on the name, number, and type of the columns in `src_tbl`. To copy only certain rows, add an appropriate `WHERE` clause. To create an empty table, use a `WHERE` clause that selects no rows:

```
CREATE TABLE dst_tbl SELECT * FROM src_tbl WHERE FALSE;
```

To copy only some of the columns, name the ones you want in the `SELECT` part of the statement. For example, if `src_tbl` contains columns `a`, `b`, `c`, and `d`, copy just `b` and `d` like this:

```
CREATE TABLE dst_tbl SELECT b, d FROM src_tbl;
```

To create columns in an order different from that in which they appear in the source table, name them in the desired order. If the source table contains columns `a`, `b`, and `c` that should appear in the destination table in the order `c`, `a`, `b`, do this:

```
CREATE TABLE dst_tbl SELECT c, a, b FROM src_tbl;
```

To create columns in the destination table in addition to those selected from the source table, provide appropriate column definitions in the `CREATE TABLE` part of the statement. The following statement creates `id` as an `AUTO_INCREMENT` column in `dst_tbl` and adds columns `a`, `b`, and `c` from `src_tbl`:

```
CREATE TABLE dst_tbl
(
  id INT NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (id)
)
SELECT a, b, c FROM src_tbl;
```

The resulting table contains four columns in the order `id`, `a`, `b`, `c`. Defined columns are assigned their default values. This means that `id`, being an `AUTO_INCREMENT` column, is assigned successive sequence numbers starting from 1 (see Recipe 13.1).

If you derive a column's values from an expression, its default name is the expression itself, which can be difficult to work with later. In this case, it's prudent to give the column a better name by providing an alias (see Recipe 3.2). Suppose that `src_tbl` contains invoice information that lists items in each invoice. The following statement generates a summary that lists each invoice named in the table and the total cost of its items, using an alias for the expression:

```
CREATE TABLE dst_tbl
SELECT inv_no, SUM(unit_cost*quantity) AS total_cost
FROM src_tbl GROUP BY inv_no;
```

`CREATE TABLE ... SELECT` is extremely convenient, but has some limitations that arise from the fact that the information available from a result set is not as extensive as what you can specify in a `CREATE TABLE` statement. For example, MySQL has no idea whether a result set column should be indexed or what its default value is. If it's important to include this information in the destination table, use the following techniques:

- To make the destination table an *exact* copy of the source table, use the cloning technique described in Recipe 4.1.
- To include indexes in the destination table, specify them explicitly. For example, if `src_tbl` has a `PRIMARY KEY` on the `id` column, and a multiple-column index on `state` and `city`, specify them for `dst_tbl` as well:

```
CREATE TABLE dst_tbl (PRIMARY KEY (id), INDEX(state,city))
SELECT * FROM src_tbl;
```

- Column attributes such as `AUTO_INCREMENT` and a column's default value are not copied to the destination table. To preserve these attributes, create the table, then use `ALTER TABLE` to apply the appropriate modifications to the column definition. For example, if `src_tbl` has an `id` column that is not only a `PRIMARY KEY` but also an `AUTO_INCREMENT` column, copy the table and modify the copy:

```
CREATE TABLE dst_tbl (PRIMARY KEY (id)) SELECT * FROM src_tbl;
ALTER TABLE dst_tbl MODIFY id INT UNSIGNED NOT NULL AUTO_INCREMENT;
```

4.3. Creating Temporary Tables

Problem

You need a table only for a short time, after which you want it to disappear automatically.

Solution

Create a table using the `TEMPORARY` keyword, and let MySQL take care of removing it.

Discussion

Some operations require a table that exists only temporarily and that should disappear when it's no longer needed. You can, of course, execute a `DROP TABLE` statement explicitly to remove a table when you're done with it. Another option is to use `CREATE TEMPORARY TABLE`. This statement is like `CREATE TABLE` but creates a transient table that disappears when your session with the server ends, if you haven't already removed it yourself. This is extremely useful behavior because MySQL drops the table for you automatically; you need not remember to do it. `TEMPORARY` can be used with the usual table-creation methods:

- Create the table from explicit column definitions:

```
CREATE TEMPORARY TABLE tbl_name (...column definitions...);
```

- Create the table from an existing table:

```
CREATE TEMPORARY TABLE new_table LIKE original_table;
```

- Create the table on the fly from a result set:

```
CREATE TEMPORARY TABLE tbl_name SELECT ... ;
```

Temporary tables are session-specific, so multiple clients can each create a temporary table having the same name without interfering with each other. This makes it easier to write applications that use transient tables because you need not ensure that the tables have unique names for each client. (For further discussion of table-naming issues, see Recipe 4.4.)

A temporary table can have the same name as a permanent table. In this case, the temporary table "hides" the permanent table for the duration of its existence, which can be useful for making a copy of a table that you can modify without affecting the original by mistake. The `DELETE` statement in the following example removes rows from a temporary `mail` table, leaving the original permanent table unaffected:

```
mysql> CREATE TEMPORARY TABLE mail SELECT * FROM mail;
mysql> SELECT COUNT(*) FROM mail;
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
mysql> DELETE FROM mail;
mysql> SELECT COUNT(*) FROM mail;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
mysql> DROP TEMPORARY TABLE mail;
mysql> SELECT COUNT(*) FROM mail;
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
```

Although temporary tables created with `CREATE TEMPORARY TABLE` have the benefits just discussed, keep the following caveats in mind:

- To reuse a temporary table within a given session, you must still drop it explicitly before re-creating it. Attempting to create a second temporary table with the same name results in an error.
- If you modify a temporary table that "hides" a permanent table with the same name, be sure to test for errors resulting from dropped connections if you use a programming interface that has reconnect capability enabled. If a client program automatically reconnects after detecting a dropped connection, modifications affect the permanent table after the reconnect, not the temporary table.

- Some APIs support persistent connections or connection pools. These prevent temporary tables from being dropped as you expect when your script ends because the connection remains open for reuse by other scripts. Your script has no control over when the connection closes. This means it can be prudent to execute the following statement prior to creating a temporary table, just in case it's still in existence from a previous execution of the script:

  ```
  DROP TEMPORARY TABLE IF EXISTS tbl_name
  ```

  The `TEMPORARY` keyword is useful here if the temporary table has already been dropped, to avoid dropping any permanent table that has the same name.

4.4. Generating Unique Table Names

Problem

You need to create a table with a name guaranteed not to exist.

Solution

If you create a `TEMPORARY` table, it doesn't matter whether a permanent table with that name exists. Otherwise, try to generate a value that is unique to your client program and incorporate it into the table name.

Discussion

MySQL is a multiple-client database server, so if a given script that creates a transient table might be invoked by several clients simultaneously, take care that multiple invocations of the script do not fight over the same table name. If the script creates tables using `CREATE TEMPORARY TABLE`, there is no problem because different clients can create temporary tables having the same name without clashing.

If you cannot or do not want to use a `TEMPORARY` table, make sure that each invocation of the script creates a uniquely named table and drops the table when it is no longer needed. To accomplish this, incorporate into the name some value guaranteed to be unique per invocation. A timestamp won't work if it's possible for two instances of a script to be invoked within the timestamp resolution. A random number may be better, but random numbers only reduce the possibility of name clashes, not eliminate it. Process ID (PID) values are a better source of unique values. PIDs are reused over time, but never for two processes at the same time, so a given PID is guaranteed to be unique among the set of currently executing processes. Use this fact to create unique table names as follows.

Perl:

```
my $tbl_name = "tmp_tbl_$$";
```

Ruby:

```
tbl_name = "tmp_tbl_" + Process.pid.to_s
```

PHP:

```
$tbl_name = "tmp_tbl_" . posix_getpid ();
```

Python:

```
import os
tbl_name = "tmp_tbl_%d" % os.getpid()
```

The PID approach should not be used in contexts such as scripts run within multi-threaded web servers in which all threads share the same process ID.

Connection identifiers are another source of unique values. The MySQL server reuses these numbers over time, but no two simultaneous connections to the server have the same ID. To get your connection ID, execute this statement and retrieve the result:

```
SELECT CONNECTION_ID();
```

It's possible to incorporate a connection ID into a table name within SQL by using prepared statements. The following example illustrates this, referring to the table name in the `CREATE TABLE` statement and a precautionary `DROP TABLE` statement:

```
SET @tbl_name = CONCAT('tmp_tbl_', CONNECTION_ID());
SET @stmt = CONCAT('DROP TABLE IF EXISTS ', @tbl_name);
PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
SET @stmt = CONCAT('CREATE TABLE ', @tbl_name, ' (i INT)');
PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Why execute the `DROP TABLE`? Because if you create a table name using an identifier such as a PID or connection ID guaranteed to be unique to a given script invocation, there may still be a chance that the table already exists if an earlier invocation of the script with the same PID created a table with the same name, but crashed before removing the table. On the other hand, any such table cannot still be in use because it will have been created by a process that is no longer running. Under these circumstances, it's safe to remove the old table if it does exist before creating the new one.

Some MySQL APIs expose the connection ID directly without requiring any statement to be executed. For example, in Perl DBI, use the `mysql_thread_id` attribute of your database handle:

```
my $tbl_name = "tmp_tbl_" . $dbh->{mysql_thread_id};
```

In Ruby DBI, do this:

```
tbl_name = "tmp_tbl_" + dbh.func(:thread_id).to_s
```

4.5. Checking or Changing a Table Storage Engine

Problem

You want to check which storage engine a table uses so that you can determine what engine capabilities are applicable. Or you need to change a table's storage engine because you realize that the capabilities of another engine are more suitable for the way you use the table.

Solution

To determine a table's storage engine, you can use any of several statements. To change the table's engine, use `ALTER TABLE` with an `ENGINE` clause.

Discussion

MySQL supports multiple storage engines, which have differing characteristics. For example, the InnoDB engine supports transactions, whereas MyISAM does not. If you need to know whether a table supports transactions, check which storage engine it uses. If the table's engine does not support transactions, you can convert the table to use a transaction-capable engine.

To determine the current engine for a table, check `INFORMATION_SCHEMA` or use the `SHOW TABLE STATUS` or `SHOW CREATE TABLE` statement. For the `mail` table, obtain engine information as follows:

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'mail';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+

mysql> SHOW TABLE STATUS LIKE 'mail'\G
*************************** 1. row ***************************
           Name: mail
         Engine: InnoDB
…

mysql> SHOW CREATE TABLE mail\G
*************************** 1. row ***************************
       Table: mail
Create Table: CREATE TABLE `mail` (
```

```
... column definitions ...
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

To change the storage engine for a table, use `ALTER TABLE` with an `ENGINE` specifier. For example, to convert the `mail` table to use the MyISAM storage engine, use this statement:

```
ALTER TABLE mail ENGINE = MyISAM;
```

Be aware that converting a large table to a different storage engine might take a long time and be expensive in terms of CPU and I/O activity.

To determine which storage engines your MySQL server supports, check the output from the `SHOW ENGINES` statement or query the `INFORMATION_SCHEMA ENGINES` table.

4.6. Copying a Table Using mysqldump

Problem

You want to copy a table or tables, either among the databases managed by a MySQL server, or from one server to another.

Solution

Use the *mysqldump* program.

Discussion

The *mysqldump* program makes a backup file that can be reloaded to re-create the original table or tables:

```
% mysqldump cookbook mail > mail.sql
```

The output file *mail.sql* consists of a `CREATE TABLE` statement to create the `mail` table and a set of `INSERT` statements to insert its rows. You can reload the file to re-create the table should the original be lost:

```
% mysql cookbook < mail.sql
```

This method also makes it easy to deal with any triggers the table has. By default, *mysqldump* writes the triggers to the dump file, so reloading the file copies the triggers along with the table with no special handling.

In addition to restoring tables, *mysqldump* can be used to make *copies* of them, by reloading the output into a different database. (If the destination database does not exist, create it first.) The following examples show some useful table-copying commands.

Copying tables within a single MySQL server

- Copy a single table to a different database:

```
% mysqldump cookbook mail > mail.sql
% mysql other_db < mail.sql
```

To dump multiple tables, name them all following the database name argument.

- Copy all tables in a database to a different database:

```
% mysqldump cookbook > cookbook.sql
% mysql other_db < cookbook.sql
```

When you name no tables after the database name, *mysqldump* dumps them all. To also include stored routines and events, add the `--routines` and `--events` options to the *mysqldump* command. (There is also a `--triggers` option, but it's unneeded because, as mentioned previously, *mysqldump* dumps triggers with their associated tables by default.)

- Copy a table, using a different name for the copy:
  - Dump the table:

```
% mysqldump cookbook mail > mail.sql
```

  - Reload the table into a different database that does *not* contain a table with that name:

```
% mysql other_db < mail.sql
```

  - Rename the table:

```
% mysql other_db
mysql> RENAME mail TO mail2;
```

Or, to move the table into another database at the same time, qualify the new name with the database name:

```
% mysql other_db
mysql> RENAME mail TO cookbook.mail2;
```

To perform a table-copying operation without an intermediary file, use a pipe to connect the *mysqldump* and *mysql* commands:

```
% mysqldump cookbook mail | mysql other_db
% mysqldump cookbook | mysql other_db
```

Copying tables between MySQL servers

The preceding commands use *mysqldump* to copy tables among the databases managed by a single MySQL server. Output from *mysqldump* can also be used to copy tables from one server to another. Suppose that you want to copy the `mail` table from the `cookbook` database on the local host to the `other_db` database on the host *otherhost.example.com*. One way to do this is to dump the output into a file:

```
% mysqldump cookbook mail > mail.sql
```

Then copy *mail.sql* to *other-host.example.com*, and run the following command there to load the table into that MySQL server's `other_db` database:

```
% mysql other_db < mail.sql
```

To accomplish this without an intermediary file, use a pipe to send the output of *mysqldump* directly over the network to the remote MySQL server. If you can connect to both servers from your local host, use this command:

```
% mysqldump cookbook mail | mysql -h other-host.example.com other_db
```

The *mysqldump* half of the command connects to the local server and writes the dump output to the pipe. The *mysql* half of the command connects to the remote MySQL server on *other-host.example.com*. It reads the pipe for input and sends each statement to the *other-host.example.com* server.

If you cannot connect directly to the remote server using *mysql* from your local host, send the dump output into a pipe that uses *ssh* to invoke *mysql* remotely on *other-host.example.com*:

```
% mysqldump cookbook mail | ssh other-host.example.com mysql other_db
```

ssh connects to *other-host.example.com* and launches *mysql* there. It then reads the *mysqldump* output from the pipe and passes it to the remote *mysql* process. *ssh* can be useful to send a dump over the network to a machine that has the MySQL port blocked by a firewall but that permits connections on the SSH port.

Regarding which table or tables to copy, similar principles apply as for local copies. To copy multiple tables over the network, name them all following the database argument of the *mysqldump* command. To copy an entire database, don't specify any table names after the database name; *mysqldump* dumps all its tables.

CHAPTER 5
Working with Strings

5.0. Introduction

Like most types of data, string values can be compared for equality or inequality or relative ordering. However, strings have additional properties to consider:

- A string can be binary or nonbinary. Binary strings are used for raw data such as images, music files, or encrypted values. Nonbinary strings are used for character data such as text and are associated with a character set and collation (sort order).
- A character set determines which characters are legal in a string. You can choose collations according to whether you need comparisons to be case sensitive or case insensitive, or to use the rules of a particular language.
- Data types for binary strings are `BINARY`, `VARBINARY`, and `BLOB`. Data types for nonbinary strings are `CHAR`, `VARCHAR`, and `TEXT`, each of which permits `CHARACTER SET` and `COLLATE` attributes.
- You can convert a binary string to a nonbinary string and vice versa, or convert a nonbinary string from one character set or collation to another.
- You can use a string in its entirety or extract substrings from it. Strings can be combined with other strings.
- You can apply pattern-matching operations to strings.
- Full-text searching is available for efficient queries on large collections of text.

This chapter discusses how to use those properties, so that you can store, retrieve, and manipulate strings according to any requirements your applications have.

Scripts to create the tables used in this chapter are located in the *tables* directory of the `recipes` distribution.

5.1. String Properties

One string property is whether it is binary or nonbinary:

- A binary string is a sequence of bytes. It can contain any type of information, such as images, MP3 files, or compressed or encrypted data. A binary string is not associated with a character set, even if you store a value such as `abc` that looks like ordinary text. Binary strings are compared byte by byte using numeric byte values.
- A nonbinary string is a sequence of characters. It stores text that has a particular character set and collation. The character set defines which characters can be stored in the string. The collation defines the character ordering, which affects comparison and sorting operations.

To see which character sets are available for nonbinary strings, use this statement:

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
…
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
…
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
…
```

The default character set in MySQL is `latin1`. If you must store characters from several languages in a single column, consider using one of the Unicode character sets (such as `utf8` or `ucs2`) because they can represent characters from multiple languages.

Some character sets contain only single-byte characters, whereas others permit multibyte characters. Some multibyte character sets contain characters of varying lengths. For others, all characters have a fixed length. For example, Unicode data can be stored using the `utf8` character set in which characters take from one to three bytes or the `ucs2` character set in which all characters take two bytes.

In MySQL, the `utf8` and `ucb2` Unicode character sets include only characters in the Basic Multilingual Plane (BMP). To use the full set of Unicode characters, including supplemental characters that lie outside the BMP, use `utf8mb4`, in which characters take from one to four bytes. Other Unicode character sets that include supplemental characters are `utf16`, `utf16le`, and `utf32`.

To determine whether a given string contains multibyte characters, use the `LENGTH()` and `CHAR_LENGTH()` functions, which return the length of a string in bytes and characters, respectively. If `LENGTH()` is greater than `CHAR_LENGTH()` for a given string, multibyte characters are present:

- The `utf8` Unicode character set has multibyte characters, but a given `utf8` string might contain only single-byte characters, as in the following example:

```
mysql> SET @s = CONVERT('abc' USING utf8);
mysql> SELECT LENGTH(@s), CHAR_LENGTH(@s);
+------------+-----------------+
| LENGTH(@s) | CHAR_LENGTH(@s) |
+------------+-----------------+
|          3 |               3 |
+------------+-----------------+
```

- For the `ucs2` Unicode character set, all characters are encoded using two bytes, even if they are single-byte characters in another character set such as `latin1`. Thus, every `ucs2` string contains multibyte characters:

```
mysql> SET @s = CONVERT('abc' USING ucs2);
mysql> SELECT LENGTH(@s), CHAR_LENGTH(@s);
+------------+-----------------+
| LENGTH(@s) | CHAR_LENGTH(@s) |
+------------+-----------------+
|          6 |               3 |
+------------+-----------------+
```

Another property of nonbinary strings is collation, which determines the sort order of characters in the character set. Use `SHOW COLLATION` to see all available collations; add a `LIKE` clause to see the collations for a particular character set:

```
mysql> SHOW COLLATION LIKE 'latin1%';
+-------------------+---------+----+---------+----------+---------+
| Collation         | Charset | Id | Default | Compiled | Sortlen |
+-------------------+---------+----+---------+----------+---------+
| latin1_german1_ci | latin1  |  5 |         | Yes      |       1 |
| latin1_swedish_ci | latin1  |  8 | Yes     | Yes      |       1 |
| latin1_danish_ci  | latin1  | 15 |         | Yes      |       1 |
| latin1_german2_ci | latin1  | 31 |         | Yes      |       2 |
| latin1_bin        | latin1  | 47 |         | Yes      |       1 |
| latin1_general_ci | latin1  | 48 |         | Yes      |       1 |
| latin1_general_cs | latin1  | 49 |         | Yes      |       1 |
| latin1_spanish_ci | latin1  | 94 |         | Yes      |       1 |
+-------------------+---------+----+---------+----------+---------+
```

In contexts where no collation is specified explicitly, strings in a given character set use the collation with `Yes` in the `Default` column. As shown, the default collation for `latin1` is `latin1_swedish_ci`. (Default collations are also displayed by `SHOW CHARACTER SET`.)

A collation can be case sensitive (`a` and `A` are different), case insensitive (`a` and `A` are the same), or binary (two characters are the same or different based on whether their numeric values are equal). A collation name ending in `_ci`, `_cs`, or `_bin` is case insensitive, case sensitive, or binary, respectively.

A binary collation provides a sort order for nonbinary strings that is something like the order for binary strings, in the sense that comparisons for binary strings and binary collations both use numeric values. The difference is that binary string comparisons are always based on single-byte units, whereas a binary collation compares nonbinary strings using *character* numeric values; depending on the character set, some of these might be multibyte values.

The following example illustrates how collation affects sort order. Suppose that a table contains a `latin1` string column and has the following rows:

```
mysql> CREATE TABLE t (c CHAR(3) CHARACTER SET latin1);
mysql> INSERT INTO t (c) VALUES('AAA'),('bbb'),('aaa'),('BBB');
mysql> SELECT c FROM t;
+------+
| c    |
+------+
| AAA  |
| bbb  |
| aaa  |
| BBB  |
+------+
```

By applying the `COLLATE` operator to the column, you can choose which collation to use for sorting and thus affect the order of the result:

- A case-insensitive collation sorts `a` and `A` together, placing them before `b` and `B`. However, for a given letter, it does not necessarily order one lettercase before another, as shown by the following result:

```
mysql> SELECT c FROM t ORDER BY c COLLATE latin1_swedish_ci;
+------+
| c    |
+------+
| AAA  |
| aaa  |
| bbb  |
| BBB  |
+------+
```

- A case-sensitive collation puts `A` and `a` before `B` and `b`, and sorts uppercase before lowercase:

```
mysql> SELECT c FROM t ORDER BY c COLLATE latin1_general_cs;
+------+
| c    |
+------+
```

```
| AAA  |
| aaa  |
| BBB  |
| bbb  |
+------+
```

- A binary collation sorts characters using their numeric values. Assuming that uppercase letters have numeric values less than those of lowercase letters, a binary collation results in the following ordering:

```
mysql> SELECT c FROM t ORDER BY c COLLATE latin1_bin;
+------+
| c    |
+------+
| AAA  |
| BBB  |
| aaa  |
| bbb  |
+------+
```

  Note that because characters in different lettercases have different numeric values, a binary collation produces a case-sensitive ordering. However, the order differs from that for the case-sensitive collation.

If you require that comparison and sorting operations use the sorting rules of a particular language, choose a language-specific collation. For example, if you store strings using the `utf8` character set, the default collation (`utf8_general_ci`) treats `ch` and `ll` as two-character strings. To use the traditional Spanish ordering that treats `ch` and `ll` as single characters that follow `c` and `l`, respectively, specify the `utf8_spanish2_ci` collation. The two collations produce different results, as shown here:

```
mysql> CREATE TABLE t (c CHAR(2) CHARACTER SET utf8);
mysql> INSERT INTO t (c) VALUES('cg'),('ch'),('ci'),('lk'),('ll'),('lm');
mysql> SELECT c FROM t ORDER BY c COLLATE utf8_general_ci;
+------+
| c    |
+------+
| cg   |
| ch   |
| ci   |
| lk   |
| ll   |
| lm   |
+------+
mysql> SELECT c FROM t ORDER BY c COLLATE utf8_spanish2_ci;
+------+
| c    |
+------+
| cg   |
| ci   |
| ch   |
```

```
| lk   |
| lm   |
| ll   |
+------+
```

5.2. Choosing a String Data Type

Problem

You want to store string data but aren't sure which is the most appropriate data type.

Solution

Choose the data type according to the characteristics of the information to be stored and how you need to use it. Consider questions such as these:

- Are the strings binary or nonbinary?
- Does case sensitivity matter?
- What is the maximum string length?
- Do you want to store fixed- or variable-length values?
- Do you need to retain trailing spaces?
- Is there a fixed set of permitted values?

Discussion

MySQL provides several binary and nonbinary string data types. These types come in pairs as shown in the following table. The maximum length is in bytes, whether the type is binary or nonbinary. For nonbinary types, the maximum number of *characters* is less for strings that contain multibyte characters:

Binary data type	Nonbinary data type	Maximum length
`BINARY`	`CHAR`	255
`VARBINARY`	`VARCHAR`	65,535
`TINYBLOB`	`TINYTEXT`	255
`BLOB`	`TEXT`	65,535
`MEDIUMBLOB`	`MEDIUMTEXT`	16,777,215
`LONGBLOB`	`LONGTEXT`	4,294,967,295

For the `BINARY` and `CHAR` data types, MySQL stores column values using a fixed width. For example, values stored in a `BINARY(10)` or `CHAR(10)` column always take 10 bytes or 10 characters, respectively. Shorter values are padded to the required length as nec-

essary when stored. For BINARY, the pad value is 0x00 (the zero-valued byte, also known as ASCII NUL). CHAR values are padded with spaces for storage and trailing spaces are stripped upon retrieval.

For VARBINARY, VARCHAR, and the BLOB and TEXT types, MySQL stores values using only as much storage as required, up to the maximum column length. No padding is added or stripped when values are stored or retrieved.

To preserve trailing pad values that are present in the original strings that are stored, use a data type for which no stripping occurs. For example, if you store character (nonbinary) strings that might end with spaces, and want to preserve them, use VARCHAR or one of the TEXT data types. The following statements illustrate the difference in trailing-space handling for CHAR and VARCHAR columns:

```
mysql> CREATE TABLE t (c1 CHAR(10), c2 VARCHAR(10));
mysql> INSERT INTO t (c1,c2) VALUES('abc      ','abc      ');
mysql> SELECT c1, c2, CHAR_LENGTH(c1), CHAR_LENGTH(c2) FROM t;
+------+------------+----------------+----------------+
| c1   | c2         | CHAR_LENGTH(c1) | CHAR_LENGTH(c2) |
+------+------------+----------------+----------------+
| abc  | abc        |               3 |              10 |
+------+------------+----------------+----------------+
```

This shows that if you store a string that contains trailing spaces into a CHAR column, they're removed when you retrieve the value.

A table can include a mix of binary and nonbinary string columns, and its nonbinary columns can use different character sets and collations. When you declare a nonbinary string column, use the CHARACTER SET and COLLATE attributes if you require a particular character set and collation. For example, if you need to store utf8 (Unicode) and sjis (Japanese) strings, you might define a table with two columns like this:

```
CREATE TABLE mytbl
(
  utf8str VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_danish_ci,
  sjisstr VARCHAR(100) CHARACTER SET sjis COLLATE sjis_japanese_ci
);
```

The CHARACTER SET and COLLATE clauses are each optional in a column definition:

- If you specify CHARACTER SET and omit COLLATE, the default collation for the character set is used.
- If you specify COLLATE and omit CHARACTER SET, the character set implied by the collation name (the first part of the name) is used. For example, utf8_danish_ci and sjis_japanese_ci imply utf8 and sjis, respectively. This means that the CHARACTER SET attributes could have been omitted from the preceding CREATE TABLE statement.

- If you omit both CHARACTER SET and COLLATE, the column is assigned the table default character set and collation. A table definition can include those attributes following the closing parenthesis at the end of the CREATE TABLE statement. If present, they apply to columns that have no explicit character set or collation of their own. If omitted, the table defaults are taken from the database defaults. You can specify the database defaults when you create the database with the CREATE DATABASE statement. The server defaults apply to the database if they are omitted.

The server default character set and collation are latin1 and latin1_swedish_ci, so strings by default use the latin1 character set and are not case sensitive. To change this, set the character_set_server and collation_server system variables at server start-up (see Recipe 22.1).

MySQL also supports ENUM and SET string types, which are used for columns that have a fixed set of permitted values. The CHARACTER SET and COLLATE attributes apply to these data types as well.

5.3. Setting the Client Connection Character Set

Problem

You're executing SQL statements or producing query results that don't use the default character set.

Solution

Use SET NAMES or an equivalent method to set your connection to the proper character set.

Discussion

When you send information back and forth between your application and the server, you may need to tell MySQL the appropriate character set. For example, the default character set is latin1, but that may not always be the proper character set to use for connections to the server. If you have Greek data, displaying it using latin1 will result in gibberish on your screen. If you use Unicode strings in the utf8 character set, latin1 might not be sufficient to represent all the characters that you might need.

To deal with this problem, configure your connection to use the appropriate character set. You have several ways to do this:

- Issue a SET NAMES statement after you connect:

```
mysql> SET NAMES 'utf8';
```

  SET NAMES permits the connection collation to be specified as well:

```
mysql> SET NAMES 'utf8' COLLATE 'utf8_general_ci';
```

- If your client program supports the `--default-character-set` option, you can use it to specify the character set at program invocation time. *mysql* is one such program. Put the option in an option file so that it takes effect each time you connect to the server:

  ```
  [mysql]
  default-character-set=utf8
  ```

- If you set the environment for your working environment using the `LANG` or `LC_ALL` environment variable on Unix, or the code page setting on Windows, MySQL client programs automatically detect which character set to use. For example, setting `LC_ALL` to `en_US.UTF-8` causes programs such as *mysql* to use `utf8`.

- Some programming interfaces provide their own method of setting the character set. For example, MySQL Connector/J for Java clients detects the character set used on the server side automatically when you connect, but you can specify a different set explicitly using the `characterEncoding` property in the connection URL. The property value should be the Java-style character-set name. To select `utf8`, you might use a connection URL like this:

  ```
  jdbc:mysql://localhost/cookbook?characterEncoding=UTF-8
  ```

  This is preferable to `SET NAMES` because Connector/J performs character-set conversion on behalf of the application, but is unaware of which character set applies if you use `SET NAMES`. Similar principles apply to programs written for other APIs. For PDO, use a `charset` option in your data source name (DSN) string (this works in PHP 5.3.6 or later):

  ```
  $dsn = "mysql:host=localhost;dbname=cookbook;charset=utf8";
  ```

  For Connector/Python, specify a `charset` connection parameter:

  ```
  conn_params = {
    "database": "cookbook",
    "host": "localhost",
    "user": "cbuser",
    "password": "cbpass",
    "charset": "utf8",
  }
  ```

  Some APIs may also provide a parameter to specify the collation.

Some character sets cannot be used as the connection character set: `ucs2`, `utf16`, `utf16le`, `utf32`.

You should also ensure that the character set used by your display device matches what you use for MySQL. Otherwise, even with MySQL handling the data properly, it might display as garbage. Suppose that you use the *mysql* program in a terminal window and that you configure MySQL to use `utf8` and store `utf8`-encoded Japanese data. If you set your terminal window to use `euc-jp` encoding, that is also Japanese, but its encoding for Japanese characters differs from `utf8`, so the data will not display as you expect. (If you use autodetection, this should not be an issue.)

In web contexts, you can include a character-set encoding in the `Content-Type:` header that precedes the web page content. See Recipe 18.1.

5.4. Writing String Literals

Problem

You need to write literal strings in SQL statements.

Solution

Learn the syntax rules that govern string values.

Discussion

You can write strings several ways:

- Enclose the text of the string within single quotes or double quotes:

  ```
  'my string'
  "my string"
  ```

  When the `ANSI_QUOTES` SQL mode is enabled, you cannot use double quotes for quoting strings: the server interprets double quote as the quoting character for identifiers such as table or column names, and not for strings (see Recipe 2.6). If you adopt the convention of always writing quoted strings using single quotes, MySQL interprets them as strings and not as identifiers regardless of the `ANSI_QUOTES` setting.

- Use hexadecimal notation. Each pair of hex digits produces one byte of the string. `abcd` can be written using any of these formats:

  ```
  0x61626364
  X'61626364'
  x'61626364'
  ```

  MySQL treats strings written using hex notation as binary strings. Not coincidentally, it's common for applications to use hex strings when constructing SQL statements that refer to binary values:

```
INSERT INTO t SET binary_col = 0xdeadbeef;
```

- To specify a character set for interpretation of a literal string, use an introducer consisting of a character-set name preceded by an underscore:

  ```
  _latin1 'abcd'
  _ucs2 'abcd'
  ```

  An introducer tells the server how to interpret the string that follows it. For `_latin1 'abcd'`, the server produces a string consisting of four single-byte characters. For `_ucs2 'abcd'`, the server produces a string consisting of two two-byte characters because `ucs2` is a double-byte character set.

To ensure that a string is a binary string or that a nonbinary string has a specific character set or collation, use the instructions for string conversion given in Recipe 5.5.

A quoted string that includes the same quote character produces a syntax error:

```
mysql> SELECT 'I'm asleep';
ERROR 1064 (42000): You have an error in your SQL syntax near 'asleep''
```

You have several ways to deal with this:

- Enclose a string containing single quotes within double quotes (assuming that `ANSI_QUOTES` is disabled), or enclose a string containing double quotes within single quotes:

  ```
  mysql> SELECT "I'm asleep", 'He said, "Boo!"';
  +------------+-----------------+
  | I'm asleep | He said, "Boo!" |
  +------------+-----------------+
  | I'm asleep | He said, "Boo!" |
  +------------+-----------------+
  ```

- To include a quote character within a string quoted by the same kind of quote, double the quote or precede it with a backslash. When MySQL reads the statement, it strips the extra quote or the backslash:

  ```
  mysql> SELECT 'I''m asleep', 'I\'m wide awake';
  +------------+----------------+
  | I'm asleep | I'm wide awake |
  +------------+----------------+
  | I'm asleep | I'm wide awake |
  +------------+----------------+
  mysql> SELECT "He said, ""Boo!""", "And I said, \"Yikes!\"";
  +-----------------+----------------------+
  | He said, "Boo!" | And I said, "Yikes!" |
  +-----------------+----------------------+
  | He said, "Boo!" | And I said, "Yikes!" |
  +-----------------+----------------------+
  ```

  A backslash turns off any special meaning of the following character, including itself. To write a literal backslash within a string, double it:

```
mysql> SELECT 'Install MySQL in C:\\mysql on Windows';
+-------------------------------------+
| Install MySQL in C:\mysql on Windows |
+-------------------------------------+
| Install MySQL in C:\mysql on Windows |
+-------------------------------------+
```

Backslash causes a temporary escape from normal string processing rules, so sequences such as `\'`, `\"`, and `\\` are called escape sequences. Others recognized by MySQL are `\b` (backspace), `\n` (newline, also called linefeed), `\r` (carriage return), `\t` (tab), and `\0` (ASCII NUL).

- Write the string as a hex value:

```
mysql> SELECT 0x49276D2061736C656570;
+------------------------+
| 0x49276D2061736C656570 |
+------------------------+
| I'm asleep             |
+------------------------+
```

See Also

If you execute SQL statements from within a program, you can refer to strings or binary values symbolically and let your programming interface take care of quoting: use the placeholder mechanism provided by the language's database-access API (see Recipe 2.5). Alternatively, load binary values such as images from files using the `LOAD_FILE()` function (see Recipe 19.6).

5.5. Checking or Changing a String's Character Set or Collation

Problem

You want to know the character set or collation of a string, or change a string to some other character set or collation.

Solution

To check a string's character set or collation, use the `CHARSET()` or `COLLATION()` function. To change its character set, use the `CONVERT()` function. To change its collation, use the `COLLATE` operator.

Discussion

For a table created as follows, you know that values stored in the column `c` have a character set of `utf8` and a collation of `utf8_danish_ci`:

```
CREATE TABLE t (c CHAR(10) CHARACTER SET utf8 COLLATE utf8_danish_ci);
```

But sometimes it's not so clear what character set or collation applies to a string. Server configuration affects literal strings and some string functions, and other string functions return values in a specific character set. Symptoms that you have the wrong character set or collation are that a collation-mismatch error occurs for a comparison operation, or a lettercase conversion doesn't work properly.

To determine a string's character set or collation, use the `CHARSET()` or `COLLATION()` function. For example, did you know that the `USER()` function returns a Unicode string?

```
mysql> SELECT USER(), CHARSET(USER()), COLLATION(USER());
+-----------------+-----------------+-------------------+
| USER()          | CHARSET(USER()) | COLLATION(USER()) |
+-----------------+-----------------+-------------------+
| cbuser@localhost | utf8           | utf8_general_ci   |
+-----------------+-----------------+-------------------+
```

String values that take their character set and collation from the current configuration may change properties if the configuration changes. This is true for literal strings:

```
mysql> SET NAMES 'latin1';
mysql> SELECT CHARSET('abc'), COLLATION('abc');
+---------------+-------------------+
| CHARSET('abc') | COLLATION('abc')  |
+---------------+-------------------+
| latin1         | latin1_swedish_ci |
+---------------+-------------------+
mysql> SET NAMES utf8 COLLATE 'utf8_bin';
mysql> SELECT CHARSET('abc'), COLLATION('abc');
+---------------+-----------------+
| CHARSET('abc') | COLLATION('abc') |
+---------------+-----------------+
| utf8           | utf8_bin        |
+---------------+-----------------+
```

For a binary string, the `CHARSET()` or `COLLATION()` functions return a value of `binary`, which means that the string is compared and sorted based on numeric byte values, not character collation values.

To convert a string from one character set to another, use the `CONVERT()` function:

```
mysql> SET @s1 = _latin1 'my string', @s2 = CONVERT(@s1 USING utf8);
mysql> SELECT CHARSET(@s1), CHARSET(@s2);
+--------------+--------------+
| CHARSET(@s1) | CHARSET(@s2) |
+--------------+--------------+
| latin1       | utf8         |
+--------------+--------------+
```

To change the collation of a string, use the `COLLATE` operator:

```
mysql> SET @s1 = _latin1 'my string', @s2 = @s1 COLLATE latin1_spanish_ci;
mysql> SELECT COLLATION(@s1), COLLATION(@s2);
+-------------------+-------------------+
| COLLATION(@s1)    | COLLATION(@s2)    |
+-------------------+-------------------+
| latin1_swedish_ci | latin1_spanish_ci |
+-------------------+-------------------+
```

The new collation must be legal for the character set of the string. For example, you can use the `utf8_general_ci` collation with `utf8` strings, but not with `latin1` strings:

```
mysql> SELECT _latin1 'abc' COLLATE utf8_bin;
ERROR 1253 (42000): COLLATION 'utf8_bin' is not valid for
CHARACTER SET 'latin1'
```

To convert both the character set and collation of a string, use `CONVERT()` to change the character set, and apply the `COLLATE` operator to the result:

```
mysql> SET @s1 = _latin1 'my string';
mysql> SET @s2 = CONVERT(@s1 USING utf8) COLLATE utf8_spanish_ci;
mysql> SELECT CHARSET(@s1), COLLATION(@s1), CHARSET(@s2), COLLATION(@s2);
+--------------+-------------------+--------------+-----------------+
| CHARSET(@s1) | COLLATION(@s1)    | CHARSET(@s2) | COLLATION(@s2)  |
+--------------+-------------------+--------------+-----------------+
| latin1       | latin1_swedish_ci | utf8         | utf8_spanish_ci |
+--------------+-------------------+--------------+-----------------+
```

The `CONVERT()` function can also convert binary strings to nonbinary strings and vice versa. To produce a binary string, use `binary`; any other character-set name produces a nonbinary string:

```
mysql> SET @s1 = _latin1 'my string';
mysql> SET @s2 = CONVERT(@s1 USING binary);
mysql> SET @s3 = CONVERT(@s2 USING utf8);
mysql> SELECT CHARSET(@s1), CHARSET(@s2), CHARSET(@s3);
+--------------+--------------+--------------+
| CHARSET(@s1) | CHARSET(@s2) | CHARSET(@s3) |
+--------------+--------------+--------------+
| latin1       | binary       | utf8         |
+--------------+--------------+--------------+
```

Alternatively, produce binary strings using the `BINARY` operator, which is equivalent to `CONVERT(`*`str`* `USING binary)`:

```
mysql> SELECT CHARSET(BINARY _latin1 'my string');
+-------------------------------------+
| CHARSET(BINARY _latin1 'my string') |
+-------------------------------------+
| binary                              |
+-------------------------------------+
```

5.6. Converting the Lettercase of a String

Problem

You want to convert a string to uppercase or lowercase.

Solution

Use the `UPPER()` or `LOWER()` function. If they don't work, you're probably trying to convert a binary string. Convert it to a nonbinary string that has a character set and collation and is subject to case mapping.

Discussion

The `UPPER()` and `LOWER()` functions convert the lettercase of a string:

```
mysql> SELECT thing, UPPER(thing), LOWER(thing) FROM limbs;
+--------------+--------------+--------------+
| thing        | UPPER(thing) | LOWER(thing) |
+--------------+--------------+--------------+
| human        | HUMAN        | human        |
| insect       | INSECT       | insect       |
| squid        | SQUID        | squid        |
| fish         | FISH         | fish         |
| centipede    | CENTIPEDE    | centipede    |
| table        | TABLE        | table        |
| armchair     | ARMCHAIR     | armchair     |
| phonograph   | PHONOGRAPH   | phonograph   |
| tripod       | TRIPOD       | tripod       |
| Peg Leg Pete | PEG LEG PETE | peg leg pete |
| space alien  | SPACE ALIEN  | space alien  |
+--------------+--------------+--------------+
```

But some strings are "stubborn" and resist lettercase conversion:

```
mysql> CREATE TABLE t (b BLOB) SELECT 'aBcD' AS b;
mysql> SELECT b, UPPER(b), LOWER(b) FROM t;
+------+----------+----------+
| b    | UPPER(b) | LOWER(b) |
+------+----------+----------+
| aBcD | aBcD     | aBcD     |
+------+----------+----------+
```

This problem occurs for strings that have a `BINARY` or `BLOB` data type. These are binary strings that have no character set or collation. Lettercase does not apply, and `UPPER()` and `LOWER()` do nothing.

To map a binary string to a given lettercase, convert it to a nonbinary string, choosing a character set that has uppercase and lowercase characters. The case-conversion functions then work as you expect because the collation provides case mapping:

```
mysql> SELECT b,
    -> UPPER(CONVERT(b USING latin1)) AS upper,
    -> LOWER(CONVERT(b USING latin1)) AS lower
    -> FROM t;
+------+-------+-------+
| b    | upper | lower |
+------+-------+-------+
| aBcD | ABCD  | abcd  |
+------+-------+-------+
```

The example uses a table column, but the same principles apply to binary string literals and string expressions.

If you're not sure whether a string expression is binary or nonbinary, use the `CHARSET()` function to find out; see Recipe 5.5.

To convert the lettercase of only part of a string, break it into pieces, convert the relevant piece, and put the pieces back together. Suppose that you want to convert only the initial character of a string to uppercase. The following expression accomplishes that:

```
CONCAT(UPPER(LEFT(str,1)),MID(str,2))
```

But it's ugly to write an expression like that each time you need it. For convenience, define a stored function:

```
mysql> CREATE FUNCTION initial_cap (s VARCHAR(255))
    -> RETURNS VARCHAR(255) DETERMINISTIC
    -> RETURN CONCAT(UPPER(LEFT(s,1)),MID(s,2));
```

Then you can capitalize initial characters more easily:

```
mysql> SELECT thing, initial_cap(thing) FROM limbs;
+-------------+--------------------+
| thing       | initial_cap(thing) |
+-------------+--------------------+
| human       | Human              |
| insect      | Insect             |
| squid       | Squid              |
| fish        | Fish               |
| centipede   | Centipede          |
| table       | Table              |
| armchair    | Armchair           |
| phonograph  | Phonograph         |
| tripod      | Tripod             |
| Peg Leg Pete | Peg Leg Pete      |
```

```
| space alien  | Space alien       |
+--------------+-------------------+
```

For more information about writing stored functions, see Chapter 9.

5.7. Controlling Case Sensitivity in String Comparisons

Problem

You want to know whether strings are equal or unequal, or which appears first in lexical order.

Solution

Use a comparison operator. But remember that strings have properties such as case sensitivity that you must take into account. A string comparison might be case sensitive when you don't want it to be, or vice versa.

Discussion

As for other data types, you can compare string values for equality, inequality, or relative ordering:

```
mysql> SELECT 'cat' = 'cat', 'cat' = 'dog', 'cat' <> 'cat', 'cat' <> 'dog';
+--------------+--------------+---------------+---------------+
| 'cat' = 'cat' | 'cat' = 'dog' | 'cat' <> 'cat' | 'cat' <> 'dog' |
+--------------+--------------+---------------+---------------+
|            1 |            0 |             0 |             1 |
+--------------+--------------+---------------+---------------+
mysql> SELECT 'cat' < 'awk', 'cat' < 'dog', 'cat' BETWEEN 'awk' AND 'eel';
+--------------+--------------+------------------------------+
| 'cat' < 'awk' | 'cat' < 'dog' | 'cat' BETWEEN 'awk' AND 'eel' |
+--------------+--------------+------------------------------+
|            0 |            1 |                            1 |
+--------------+--------------+------------------------------+
```

However, comparison and sorting properties of strings are subject to complications that don't apply to other types of data. For example, sometimes you must ensure that a string comparison is case sensitive that would not otherwise be, or vice versa. This section describes how to do that.

String comparison properties depend on whether the operands are binary or nonbinary strings:

- A binary string is a sequence of bytes and is compared using numeric byte values. Lettercase has no meaning. However, because letters in different cases have different byte values, comparisons of binary strings effectively are case sensitive. (That is, a

and A are unequal.) To compare binary strings such that lettercase does not matter, convert them to nonbinary strings that have a case-insensitive collation.

- A nonbinary string is a sequence of characters and is compared in character units. (Depending on the character set, some characters might have multiple bytes.) The string has a character set that defines the legal characters and a collation that defines their sort order. The collation also determines whether to consider characters in different lettercases the same in comparisons. If the collation is case sensitive, and you want a case-insensitive collation (or vice versa), convert the strings to use a collation with the desired case-comparison properties.

By default, strings have a character set of `latin1` and a collation of `latin1_swedish_ci` unless you reconfigure the server (see Recipe 22.1). This results in case-insensitive string comparisons.

The following example shows how two binary strings that compare as unequal can be handled so that they are equal when compared as case-insensitive nonbinary strings:

```
mysql> SET @s1 = BINARY 'cat', @s2 = BINARY 'CAT';
mysql> SELECT @s1 = @s2;
+-----------+
| @s1 = @s2 |
+-----------+
|         0 |
+-----------+
mysql> SET @s1 = CONVERT(@s1 USING latin1) COLLATE latin1_swedish_ci;
mysql> SET @s2 = CONVERT(@s2 USING latin1) COLLATE latin1_swedish_ci;
mysql> SELECT @s1 = @s2;
+-----------+
| @s1 = @s2 |
+-----------+
|         1 |
+-----------+
```

In this case, because `latin1_swedish_ci` is the default collation for `latin1`, you can omit the `COLLATE` operator:

```
mysql> SET @s1 = CONVERT(@s1 USING latin1);
mysql> SET @s2 = CONVERT(@s2 USING latin1);
mysql> SELECT @s1 = @s2;
+-----------+
| @s1 = @s2 |
+-----------+
|         1 |
+-----------+
```

The next example shows how to compare, in case-sensitive fashion, two strings that are not case sensitive:

```
mysql> SET @s1 = _latin1 'cat', @s2 = _latin1 'CAT';
mysql> SELECT @s1 = @s2;
```

```
+-----------+
| @s1 = @s2 |
+-----------+
|         1 |
+-----------+
mysql> SELECT @s1 COLLATE latin1_general_cs = @s2 COLLATE latin1_general_cs
    ->   AS '@s1 = @s2';
+-----------+
| @s1 = @s2 |
+-----------+
|         0 |
+-----------+
```

If you compare a binary string with a nonbinary string, the comparison treats both operands as binary strings:

```
mysql> SELECT _latin1 'cat' = BINARY 'CAT';
+------------------------------+
| _latin1 'cat' = BINARY 'CAT' |
+------------------------------+
|                            0 |
+------------------------------+
```

Thus, to compare two nonbinary strings as binary strings, apply the `BINARY` operator to either one when comparing them:

```
mysql> SET @s1 = _latin1 'cat', @s2 = _latin1 'CAT';
mysql> SELECT @s1 = @s2, BINARY @s1 = @s2, @s1 = BINARY @s2;
+-----------+-----------------+-----------------+
| @s1 = @s2 | BINARY @s1 = @s2 | @s1 = BINARY @s2 |
+-----------+-----------------+-----------------+
|         1 |               0 |               0 |
+-----------+-----------------+-----------------+
```

If you find that you've declared a column using a type not suited to the kind of comparisons for which you typically use it, use `ALTER TABLE` to change the type. Suppose that this table stores news articles:

```
CREATE TABLE news
(
  id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
  article BLOB,
  PRIMARY KEY (id)
);
```

Here the `article` column is declared as a `BLOB`. That is a binary string type, so comparisons of text stored in the column are made without regard to character set. (In effect, they are case sensitive.) If that's not what you want, use `ALTER TABLE` to convert the column to a nonbinary type that has a case-insensitive collation:

```
ALTER TABLE news
  MODIFY article TEXT CHARACTER SET utf8 COLLATE utf8_general_ci;
```

5.8. Pattern Matching with SQL Patterns

Problem

You want to perform a pattern match, not a literal comparison.

Solution

Use the `LIKE` operator and an SQL pattern, described in this section. Or use a regular-expression pattern match, described in Recipe 5.9.

Discussion

Patterns are strings that contain special characters known as metacharacters because they stand for something other than themselves. MySQL provides two kinds of pattern matching. One is based on SQL patterns and the other on regular expressions. SQL patterns are more standard among different database systems, but regular expressions are more powerful. The two kinds of pattern match use different operators and different metacharacters. This section describes SQL patterns. Recipe 5.9 describes regular expressions.

The example here uses a table named `metal` that contains the following rows:

```
+----------+
| name     |
+----------+
| gold     |
| iron     |
| lead     |
| mercury  |
| platinum |
| tin      |
+----------+
```

SQL pattern matching uses the `LIKE` and `NOT LIKE` operators rather than `=` and `<>` to perform matching against a pattern string. Patterns may contain two special metacharacters: `_` matches any single character, and `%` matches any sequence of characters, including the empty string. You can use these characters to create patterns that match a variety of values:

- Strings that begin with a particular substring:

```
mysql> SELECT name FROM metal WHERE name LIKE 'me%';
+---------+
| name    |
+---------+
| mercury |
+---------+
```

- Strings that end with a particular substring:

```
mysql> SELECT name FROM metal WHERE name LIKE '%d';
+------+
| name |
+------+
| gold |
| lead |
+------+
```

- Strings that contain a particular substring at any position:

```
mysql> SELECT name FROM metal WHERE name LIKE '%in%';
+----------+
| name     |
+----------+
| platinum |
| tin      |
+----------+
```

- Strings that contain a substring at a specific position (the pattern matches only if `at` occurs at the third position of the `name` column):

```
mysql> SELECT name FROM metal where name LIKE '__at%';
+----------+
| name     |
+----------+
| platinum |
+----------+
```

An SQL pattern matches successfully only if it matches the entire comparison value. Of the following two pattern matches, only the second succeeds:

```
'abc' LIKE 'b'
'abc' LIKE '%b%'
```

To reverse the sense of a pattern match, use `NOT LIKE`. The following statement finds strings that contain no `i` characters:

```
mysql> SELECT name FROM metal WHERE name NOT LIKE '%i%';
+---------+
| name    |
+---------+
| gold    |
| lead    |
| mercury |
+---------+
```

SQL patterns do not match `NULL` values. This is true both for `LIKE` and for `NOT LIKE`:

```
mysql> SELECT NULL LIKE '%', NULL NOT LIKE '%';
+---------------+-------------------+
| NULL LIKE '%' | NULL NOT LIKE '%' |
+---------------+-------------------+
|          NULL |              NULL |
+---------------+-------------------+
```

In some cases, pattern matches are equivalent to substring comparisons. For example, using patterns to find strings at one end or the other of a string is like using `LEFT()` or `RIGHT()`, as shown in the following table:

Pattern match	Substring comparison
str LIKE 'abc%'	LEFT(*str*,3) = 'abc'
str LIKE '%abc'	RIGHT(*str*,3) = 'abc'

If you're matching against a column that is indexed and you have a choice of using a pattern or an equivalent `LEFT()` expression, you'll likely find the pattern match to be faster. MySQL can use the index to narrow the search for a pattern that begins with a literal string. With `LEFT()`, it cannot.

Case sensitivity of a pattern match is like that of a string comparison. That is, it depends on whether the operands are binary or nonbinary strings, and for nonbinary strings, it depends on their collation. See Recipe 5.7 for discussion of how these factors apply to comparisons.

Using Patterns with Nonstring Values

Unlike some other database systems, MySQL permits pattern matches to be applied to nonstring values such as numbers or dates, which can sometimes be useful. The following table shows some ways to test a `DATE` value `d` using function calls that extract date parts and using the equivalent pattern matches. The pairs of expressions are true for dates occurring in the year 1976, in the month of April, or on the first day of the month:

Function value test	Pattern match test
YEAR(d) = 1976	d LIKE '1976-%'
MONTH(d) = 4	d LIKE '%-04-%'
DAYOFMONTH(d) = 1	d LIKE '%-01'

5.9. Pattern Matching with Regular Expressions

Problem

You want to perform a pattern match, not a literal comparison.

Solution

Use the REGEXP operator and a regular expression pattern, described in this section. Or use an SQL pattern, described in Recipe 5.8.

Discussion

SQL patterns (see Recipe 5.8) are likely to be implemented by other database systems, so they're reasonably portable beyond MySQL. On the other hand, they're somewhat limited. For example, you can easily write an SQL pattern `%abc%` to find strings that contain `abc`, but you cannot write a single SQL pattern to identify strings that contain any of the characters `a`, `b`, or `c`. Nor can you match string content based on character types such as letters or digits. For such operations, MySQL supports another type of pattern-matching operation based on regular expressions and the REGEXP operator (or NOT REGEXP to reverse the sense of the match). REGEXP matching uses the pattern elements shown in the following table:

Pattern	What the pattern matches
^	Beginning of string
$	End of string
.	Any single character
[...]	Any character listed between the square brackets
[^...]	Any character not listed between the square brackets
p1|*p2*|*p3*	Alternation; matches any of the patterns *p1*, *p2*, or *p3*
*	Zero or more instances of preceding element
+	One or more instances of preceding element
{*n*}	*n* instances of preceding element
{*m*,*n*}	*m* through *n* instances of preceding element

You may already be familiar with these regular expression pattern characters; many of them are the same as those used by *vi*, *grep*, *sed*, and other Unix utilities that support regular expressions. Most of them are used also in the regular expressions understood by programming languages. (For discussion of pattern matching in programs for data validation and transformation, see Chapter 12.)

Recipe 5.8 shows how to use SQL patterns to match substrings at the beginning or end of a string, or at an arbitrary or specific position within a string. You can do the same things with regular expressions:

- Strings that begin with a particular substring:

```
mysql> SELECT name FROM metal WHERE name REGEXP '^me';
+---------+
| name    |
```

```
+---------+
| mercury |
+---------+
```

- Strings that end with a particular substring:

```
mysql> SELECT name FROM metal WHERE name REGEXP 'd$';
+------+
| name |
+------+
| gold |
| lead |
+------+
```

- Strings that contain a particular substring at any position:

```
mysql> SELECT name FROM metal WHERE name REGEXP 'in';
+----------+
| name     |
+----------+
| platinum |
| tin      |
+----------+
```

- Strings that contain a particular substring at a specific position:

```
mysql> SELECT name FROM metal WHERE name REGEXP '^..at';
+----------+
| name     |
+----------+
| platinum |
+----------+
```

In addition, regular expressions have other capabilities and can perform matches that SQL patterns cannot. For example, regular expressions can contain character classes, which match any character in the class:

- To write a character class, use square brackets and list the characters you want the class to match inside the brackets. Thus, the pattern `[abc]` matches `a`, `b`, or `c`.
- Classes can indicate ranges of characters; use a dash between the beginning and end of the range. `[a-z]` matches any letter, `[0-9]` matches digits, and `[a-z0-9]` matches letters or digits.
- To negate a character class ("match any character but these"), begin the list with a `^` character. For example, `[^0-9]` matches anything but digits.

MySQL's regular-expression capabilities also support POSIX character classes. These match specific character sets, as described in the following table:

POSIX class	What the class matches
`[:alnum:]`	Alphabetic and numeric characters
`[:alpha:]`	Alphabetic characters
`[:blank:]`	Whitespace (space or tab characters)
`[:cntrl:]`	Control characters
`[:digit:]`	Digits
`[:graph:]`	Graphic (nonblank) characters
`[:lower:]`	Lowercase alphabetic characters
`[:print:]`	Graphic or space characters
`[:punct:]`	Punctuation characters
`[:space:]`	Space, tab, newline, carriage return
`[:upper:]`	Uppercase alphabetic characters
`[:xdigit:]`	Hexadecimal digits (`0-9`, `a-f`, `A-F`)

POSIX classes are intended for use within character classes, so use them within square brackets. The following expression matches values that contain any hexadecimal digit character:

```
mysql> SELECT name, name REGEXP '[[:xdigit:]]' FROM metal;
+----------+----------------------------+
| name     | name REGEXP '[[:xdigit:]]' |
+----------+----------------------------+
| gold     |                          1 |
| iron     |                          0 |
| lead     |                          1 |
| mercury  |                          1 |
| platinum |                          1 |
| tin      |                          0 |
+----------+----------------------------+
```

Regular expressions can specify alternations using this syntax:

```
alternative1|alternative2|...
```

An alternation is similar to a character class in the sense that it matches if any of the alternatives match. But unlike a character class, the alternatives are not limited to single characters. They can be multiple-character strings or even patterns. The following alternation matches strings that begin with a vowel or end with `er`:

```
mysql> SELECT name FROM metal WHERE name REGEXP '^[aeiou]|d$';
+------+
| name |
+------+
| gold |
| iron |
| lead |
+------+
```

Parentheses can be used to group alternations. For example, to match strings that consist entirely of digits or entirely of letters, you might try this pattern, using an alternation:

```
mysql> SELECT '0m' REGEXP '^[[:digit:]]+|[[:alpha:]]+$';
+-----------------------------------------+
| '0m' REGEXP '^[[:digit:]]+|[[:alpha:]]+$' |
+-----------------------------------------+
|                                       1 |
+-----------------------------------------+
```

However, as the query result shows, the pattern doesn't work. That's because the `^` groups with the first alternative, and the `$` groups with the second alternative. So the pattern actually matches strings that begin with one or more digits, or strings that end with one or more letters. If you group the alternatives within parentheses, the `^` and `$` apply to both of them, and the pattern acts as you expect:

```
mysql> SELECT '0m' REGEXP '^([[:digit:]]+|[[:alpha:]]+)$';
+-------------------------------------------+
| '0m' REGEXP '^([[:digit:]]+|[[:alpha:]]+)$' |
+-------------------------------------------+
|                                         0 |
+-------------------------------------------+
```

Unlike SQL pattern matches, which are successful only if the pattern matches the entire comparison value, regular expressions are successful if the pattern matches anywhere within the value. The following two pattern matches are equivalent in the sense that each one succeeds only for strings that contain a `b` character, but the first is more efficient because the pattern is simpler:

```
'abc' REGEXP 'b'
'abc' REGEXP '^.*b.*$'
```

Regular expressions do not match `NULL` values. This is true both for `REGEXP` and for `NOT REGEXP`:

```
mysql> SELECT NULL REGEXP '.*', NULL NOT REGEXP '.*';
+-----------------+----------------------+
| NULL REGEXP '.*' | NULL NOT REGEXP '.*' |
+-----------------+----------------------+
|            NULL |                 NULL |
+-----------------+----------------------+
```

Because a regular expression matches a string if the pattern is found anywhere in the string, you must take care not to inadvertently specify a pattern that matches the empty string. If you do, it matches any non-`NULL` value. For example, the pattern `a*` matches any number of `a` characters, even none. If your goal is to match only strings containing nonempty sequences of `a` characters, use `a+` instead. The `+` requires one or more instances of the preceding pattern element for a match.

As with SQL pattern matches performed using `LIKE`, regular-expression matches performed with `REGEXP` sometimes are equivalent to substring comparisons. As shown in

the following table, the ^ and $ metacharacters serve much the same purpose as LEFT() or RIGHT(), at least if you're looking for literal strings:

Pattern match	Substring comparison
str REGEXP '^abc'	LEFT(*str*,3) = 'abc'
str REGEXP 'abc$'	RIGHT(*str*,3) = 'abc'

For nonliteral patterns, it's typically not possible to construct an equivalent substring comparison. For example, to match strings that begin with any nonempty sequence of digits, use this pattern match:

```
str REGEXP '^[0-9]+'
```

That is something that LEFT() cannot do (and neither can LIKE, for that matter).

Case sensitivity of a regular-expression match is like that of a string comparison. That is, it depends on whether the operands are binary or nonbinary strings, and for nonbinary strings, it depends on their collation. See Recipe 5.7 for discussion of how these factors apply to comparisons.

A limitation of regular-expression (REGEXP) matching compared to SQL pattern (LIKE) matching is that REGEXP works only for single-byte character sets. Don't expect it to work with multibyte character sets such as utf8 or sjis.

5.10. Breaking Apart or Combining Strings

Problem

You want to extract a piece of a string or combine strings to form a larger string.

Solution

To obtain a piece of a string, use a substring-extraction function. To combine strings, use CONCAT().

Discussion

You can break apart strings by using appropriate substring-extraction functions. For example, LEFT(), MID(), and RIGHT() extract substrings from the left, middle, or right part of a string:

```
mysql> SET @date = '2015-07-21';
mysql> SELECT @date, LEFT(@date,4) AS year,
    -> MID(@date,6,2) AS month, RIGHT(@date,2) AS day;
+------------+------+-------+------+
```

```
| @date      | year | month | day  |
+------------+------+-------+------+
| 2015-07-21 | 2015 | 07    | 21   |
+------------+------+-------+------+
```

For `LEFT()` and `RIGHT()`, the second argument indicates how many characters to return from the left or right end of the string. For `MID()`, the second argument is the starting position of the substring you want (beginning from 1), and the third argument indicates how many characters to return.

The `SUBSTRING()` function takes a string and a starting position, returning everything to the right of the position. `MID()` acts the same way if you omit its third argument because `MID()` is actually a synonym for `SUBSTRING()`:

```
mysql> SET @date = '2015-07-21';
mysql> SELECT @date, SUBSTRING(@date,6), MID(@date,6);
+------------+--------------------+--------------+
| @date      | SUBSTRING(@date,6) | MID(@date,6) |
+------------+--------------------+--------------+
| 2015-07-21 | 07-21              | 07-21        |
+------------+--------------------+--------------+
```

Use `SUBSTRING_INDEX(`*`str`*`,`*`c`*`,`*`n`*`)` to return everything to the right or left of a given character. It searches into a string *`str`* for the *`n`*-th occurrence of the character *`c`* and returns everything to its left. If *`n`* is negative, the search for *`c`* starts from the right and returns everything to the right of the character:

```
mysql> SET @email = 'postmaster@example.com';
mysql> SELECT @email,
    -> SUBSTRING_INDEX(@email,'@',1) AS user,
    -> SUBSTRING_INDEX(@email,'@',-1) AS host;
+-----------------------+------------+-------------+
| @email                | user       | host        |
+-----------------------+------------+-------------+
| postmaster@example.com | postmaster | example.com |
+-----------------------+------------+-------------+
```

If there is no *`n`*-th occurrence of the character, `SUBSTRING_INDEX()` returns the entire string. `SUBSTRING_INDEX()` is case sensitive.

You can use substrings for purposes other than display, such as to perform comparisons. The following statement finds metal names having a first letter that lies in the last half of the alphabet:

```
mysql> SELECT name from metal WHERE LEFT(name,1) >= 'n';
+----------+
| name     |
+----------+
| platinum |
| tin      |
+----------+
```

To combine rather than pull apart strings, use the `CONCAT()` function. It concatenates its arguments and returns the result:

```
mysql> SELECT CONCAT(name,' ends in "d": ',IF(RIGHT(name,1)='d','YES','NO'))
    -> AS 'ends in "d"?'
    -> FROM metal;
+--------------------------+
| ends in "d"?             |
+--------------------------+
| gold ends in "d": YES    |
| iron ends in "d": NO     |
| lead ends in "d": YES    |
| mercury ends in "d": NO  |
| platinum ends in "d": NO |
| tin ends in "d": NO      |
+--------------------------+
```

Concatenation can be useful for modifying column values "in place." For example, the following `UPDATE` statement adds a string to the end of each `name` value in the `metal` table:

```
mysql> UPDATE metal SET name = CONCAT(name,'ide');
mysql> SELECT name FROM metal;
+-------------+
| name        |
+-------------+
| goldide     |
| ironide     |
| leadide     |
| mercuryide  |
| platinumide |
| tinide      |
+-------------+
```

To undo the operation, strip the last three characters (the `CHAR_LENGTH()` function returns the length of a string in characters):

```
mysql> UPDATE metal SET name = LEFT(name,CHAR_LENGTH(name)-3);
mysql> SELECT name FROM metal;
+----------+
| name     |
+----------+
| gold     |
| iron     |
| lead     |
| mercury  |
| platinum |
| tin      |
+----------+
```

The concept of modifying a column in place can be applied to `ENUM` or `SET` values as well, which usually can be treated as string values even though they are stored internally

as numbers. For example, to concatenate a SET element to an existing SET column, use CONCAT() to add the new value to the existing value, preceded by a comma. But remember to account for the possibility that the existing value might be NULL. In that case, set the column value equal to the new element, without the leading comma:

```
UPDATE tbl_name
SET set_col = IF(set_col IS NULL,val,CONCAT(set_col,',',val));
```

5.11. Searching for Substrings

Problem

You want to know whether a given string occurs within another string.

Solution

Use LOCATE() or a pattern match.

Discussion

The LOCATE() function takes two arguments representing the substring that you're looking for and the string in which to look for it. The return value is the position at which the substring occurs, or 0 if it's not present. An optional third argument may be given to indicate the position within the string at which to start looking.

```
mysql> SELECT name, LOCATE('in',name), LOCATE('in',name,3) FROM metal;
+----------+-------------------+--------------------+
| name     | LOCATE('in',name) | LOCATE('in',name,3) |
+----------+-------------------+--------------------+
| gold     |                 0 |                  0 |
| iron     |                 0 |                  0 |
| lead     |                 0 |                  0 |
| mercury  |                 0 |                  0 |
| platinum |                 5 |                  5 |
| tin      |                 2 |                  0 |
+----------+-------------------+--------------------+
```

To determine only whether the substring is present if you don't care about its position, an alternative is to use LIKE or REGEXP:

```
mysql> SELECT name, name LIKE '%in%', name REGEXP 'in' FROM metal;
+----------+-----------------+------------------+
| name     | name LIKE '%in%' | name REGEXP 'in' |
+----------+-----------------+------------------+
| gold     |               0 |                0 |
| iron     |               0 |                0 |
| lead     |               0 |                0 |
| mercury  |               0 |                0 |
| platinum |               1 |                1 |
```

```
| tin      |               1 |                 1 |
+----------+-----------------+-------------------+
```

`LOCATE()`, `LIKE`, and `REGEXP` use the collation of their arguments to determine whether the search is case sensitive. Recipes 5.5 and 5.7 discuss changing the argument comparison properties if you want to change the search behavior.

5.12. Using Full-Text Searches

Problem

You want to search a lot of text.

Solution

Use a `FULLTEXT` index.

Discussion

Pattern matches enable you to look through any number of rows, but as the amount of text goes up, the match operation can become quite slow. It's also a common task to search for the same text in several string columns, but with pattern matching, that results in unwieldy queries:

```
SELECT * from tbl_name
WHERE col1 LIKE 'pat' OR col2 LIKE 'pat' OR col3 LIKE 'pat' ...
```

A useful alternative is full-text searching, which is designed for looking through large amounts of text and can search multiple columns simultaneously. To use this capability, add a `FULLTEXT` index to your table, and then use the `MATCH` operator to look for strings in the indexed column or columns. `FULLTEXT` indexing can be used with MyISAM tables (or, as of MySQL 5.6, InnoDB tables) for nonbinary string data types (`CHAR`, `VARCHAR`, or `TEXT`).

Full-text searching is best illustrated with a reasonably good-sized body of text. If you don't have a sample dataset, you can find several repositories of freely available electronic text on the Internet. For the examples here, the one I've chosen is the complete text of the King James Version of the Bible (KJV), which is both relatively large and nicely structured by book, chapter, and verse. Because of its size, this dataset is not included with the `recipes` distribution, but is available separately as the `mcb-kjv` distribution at the *MySQL Cookbook* website (see the Preface). The `mcb-kjv` distribution includes a file named *kjv.txt* that contains the verse records. Some sample records look like this:

```
O   Genesis 1   1   1   In the beginning God created the heaven and the earth.
O   Exodus  2   20  13  Thou shalt not kill.
N   Luke    42  17  32  Remember Lot's wife.
```

Each record contains the following fields:

- Book section (`O` or `N`, signifying Old or New Testament)
- Book name and corresponding book number, from 1 to 66
- Chapter and verse numbers
- Text of the verse

To import the records into MySQL, create a table named `kjv` that looks like this:

```
CREATE TABLE kjv
(
  bsect ENUM('O','N') NOT NULL,    # book section (testament)
  bname VARCHAR(20) NOT NULL,      # book name
  bnum  TINYINT UNSIGNED NOT NULL, # book number
  cnum  TINYINT UNSIGNED NOT NULL, # chapter number
  vnum  TINYINT UNSIGNED NOT NULL, # verse number
  vtext TEXT NOT NULL,             # text of verse
  FULLTEXT (vtext)                 # full-text index
) ENGINE = MyISAM;                 # can be InnoDB for MySQL 5.6+
```

The table has a `FULLTEXT` index to enable its use in full-text searching. It also uses the MyISAM storage engine. If you have MySQL 5.6 or higher and want to use InnoDB instead, modify the `ENGINE` clause to `ENGINE = InnoDB`.

After creating the `kjv` table, load the *kjv.txt* file into it using this statement:

```
mysql> LOAD DATA LOCAL INFILE 'kjv.txt' INTO TABLE kjv;
```

You'll notice that the `kjv` table contains columns both for book names (Genesis, Exodus, ...) and for book numbers (1, 2, ...). The names and numbers have a fixed correspondence, and one can be derived from the other—a redundancy that means the table is not in normal form. It's possible to eliminate the redundancy by storing just the book numbers (which take less space than the names), and then producing the names when necessary in query results by joining the numbers to a mapping table that associates each book number with the corresponding name. But I want to avoid using joins at this point. Thus, the table includes book names so search results can be interpreted more easily, and numbers so the results can be sorted easily into book order.

To perform a search using the `FULLTEXT` index, use `MATCH()` to name the indexed column and `AGAINST()` to specify what text to look for. For example, you might wonder, "How many times does the name Hadoram occur?" To answer that question, search the `vtext` column using this statement:

```
mysql> SELECT COUNT(*) from kjv WHERE MATCH(vtext) AGAINST('Hadoram');
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

To find out what those verses are, select the columns you want to see (the example here truncates the `vtext` column and uses `\G` so the results better fit the page):

```
mysql> SELECT bname, cnum, vnum, LEFT(vtext,65) AS vtext
    -> FROM kjv WHERE MATCH(vtext) AGAINST('Hadoram')\G
*************************** 1. row ***************************
bname: Genesis
 cnum: 10
 vnum: 27
vtext: And Hadoram, and Uzal, and Diklah,
*************************** 2. row ***************************
bname: 1 Chronicles
 cnum: 1
 vnum: 21
vtext: Hadoram also, and Uzal, and Diklah,
*************************** 3. row ***************************
bname: 1 Chronicles
 cnum: 18
 vnum: 10
vtext: He sent Hadoram his son to king David, to inquire of his welfare,
*************************** 4. row ***************************
bname: 2 Chronicles
 cnum: 10
 vnum: 18
vtext: Then king Rehoboam sent Hadoram that was over the tribute; and th
```

The results may come out in book, chapter, and verse number order, but that's just coincidence. By default, full-text searches compute a relevance ranking and use it for sorting. To make sure a search result is sorted the way you want, add an explicit `ORDER BY` clause:

```
SELECT bname, cnum, vnum, vtext
FROM kjv WHERE MATCH(vtext) AGAINST('search string')
ORDER BY bnum, cnum, vnum;
```

To see the relevance ranking, repeat the `MATCH()` … `AGAINST()` expression in the output column list.

To narrow the search further, include additional criteria. The following queries perform progressively more specific searches to determine how often the name Abraham occurs in the entire KJV, the New Testament, the Book of Hebrews, and Chapter 11 of Hebrews:

```
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham');
+----------+
| COUNT(*) |
+----------+
|      229 |
+----------+
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham')
    -> AND bsect = 'N';
```

```
+----------+
| COUNT(*) |
+----------+
|       69 |
+----------+
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham')
    -> AND bname = 'Hebrews';
+----------+
| COUNT(*) |
+----------+
|       10 |
+----------+
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham')
    -> AND bname = 'Hebrews' AND cnum = 11;
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

If you expect to use search criteria frequently that include other non-`FULLTEXT` columns, add regular indexes to those columns so that queries perform better. For example, to index the book, chapter, and verse columns, do this:

```
mysql> ALTER TABLE kjv ADD INDEX (bnum), ADD INDEX (cnum), ADD INDEX (vnum);
```

Search strings in full-text queries can include more than one word, and you might suppose that adding words would make a search more specific. But in fact that widens it because a full-text search returns rows that contain any of the words. In effect, the query performs an `OR` search for any of the words. The following queries illustrate this; they identify successively larger numbers of verses as additional search words are added:

```
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham');
+----------+
| COUNT(*) |
+----------+
|      229 |
+----------+
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham Sarah');
+----------+
| COUNT(*) |
+----------+
|      243 |
+----------+
mysql> SELECT COUNT(*) from kjv
    -> WHERE MATCH(vtext) AGAINST('Abraham Sarah Ishmael Isaac');
+----------+
| COUNT(*) |
```

```
+----------+
|      334 |
+----------+
```

To perform a search for which each word in the search string must be present, see Recipe 5.14.

To use full-text searches that look through multiple columns simultaneously, name all the columns when you construct the FULLTEXT index:

```
ALTER TABLE tbl_name ADD FULLTEXT (col1, col2, col3);
```

To issue a search query that uses the index, name those same columns in the MATCH() list:

```
SELECT ... FROM tbl_name
WHERE MATCH(col1, col2, col3) AGAINST('search string');
```

You need one such FULLTEXT index for each distinct combination of columns that you want to search.

See Also

FULLTEXT indexes provide a quick-and-easy way to set up a basic search engine. One way to use this capability is to provide a web-based interface to the indexed text. This book's website (see the Preface) includes a simple web-based KJV search page that demonstrates this. You can use it as the basis for your own search engine that operates on a different repository of text. The search script, *kjv.pl*, is included in the mcb-kjv distribution.

5.13. Using a Full-Text Search with Short Words

Problem

Full-text searches for short words return no rows.

Solution

Change the indexing engine's minimum word length parameter.

Discussion

In a text like the KJV, certain words have special significance, such as "God" and "sin." However, if your kjv table uses the MyISAM storage engine and you perform full-text searches for those words, you'll observe a curious phenomenon—both words appear to be missing from the text entirely:

```
mysql> SELECT COUNT(*) FROM kjv WHERE MATCH(vtext) AGAINST('God');
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
mysql> SELECT COUNT(*) FROM kjv WHERE MATCH(vtext) AGAINST('sin');
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
```

One property of the indexing engine is that it ignores words that are "too common" (that is, words that occur in more than half the rows). This eliminates words such as "the" or "and" from the index, but that's not what is going on here. You can verify that by counting the total number of rows, and by using SQL pattern matches to count the number of rows containing each word (see Recipe 8.1 regarding the use of `COUNT()` to produce multiple counts from the same set of values):

```
mysql> SELECT COUNT(*) AS 'total verses',
    -> COUNT(IF(vtext LIKE '%God%',1,NULL)) AS 'verses containing "God"',
    -> COUNT(IF(vtext LIKE '%sin%',1,NULL)) AS 'verses containing "sin"'
    -> FROM kjv;
+--------------+------------------------+------------------------+
| total verses | verses containing "God" | verses containing "sin" |
+--------------+------------------------+------------------------+
|        31102 |                   4117 |                   1292 |
+--------------+------------------------+------------------------+
```

Neither word is present in more than half the verses, so sheer frequency of occurrence doesn't account for the failure of a full-text search to find them. What's really happening is that, by default, the MyISAM full-text indexing engine doesn't include words less than four characters long. The minimum word length is a configurable parameter; to change it, set the `ft_min_word_len` system variable. For example, to tell the indexing engine to include words as short as three characters, add a line to the `[mysqld]` group of the */etc/my.cnf* file (or whatever option file you use for server settings):

```
[mysqld]
ft_min_word_len=3
```

After making this change, restart the server. Next, rebuild the `FULLTEXT` index to take advantage of the new setting:

```
mysql> REPAIR TABLE kjv QUICK;
```

(You should also use `REPAIR TABLE` to rebuild the indexes for all other MyISAM tables that have `FULLTEXT` indexes.)

Finally, try the new index to verify that it includes shorter words:

```
mysql> SELECT COUNT(*) FROM kjv WHERE MATCH(vtext) AGAINST('God');
+----------+
| COUNT(*) |
+----------+
|     3892 |
+----------+
mysql> SELECT COUNT(*) FROM kjv WHERE MATCH(vtext) AGAINST('sin');
+----------+
| COUNT(*) |
+----------+
|      389 |
+----------+
```

That's better!

But why do the `MATCH()` queries find 3,892 and 389 rows, whereas the earlier `LIKE` queries find 4,117 and 1,292 rows? That's because the `LIKE` patterns match substrings and the full-text search performed by `MATCH()` matches whole words.

If your `kjv` table uses the InnoDB storage engine, you won't see the behavior just described because the default word length is 3 to begin with. However, specific values aside, similar principles apply:

- There is a minimum word length parameter, `innodb_ft_min_token_size` in this case.
- You can set that parameter at startup. If you change it from its previous value, you should rebuild all InnoDB table `FULLTEXT` indexes. InnoDB does not support `REPAIR TABLE`, but you can drop and re-create each index. For example:

```
mysql> ALTER TABLE kjv DROP INDEX vtext, ADD FULLTEXT (vtext);
```

5.14. Requiring or Prohibiting Full-Text Search Words

Problem

You want to require or prohibit specific words in a full-text search.

Solution

Use a Boolean mode search.

Discussion

Normally, full-text searches return rows that contain any of the words in the search string, even if some of them are missing. For example, the following statement finds rows that contain either of the names David or Goliath:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('David Goliath');
+----------+
| COUNT(*) |
+----------+
|      898 |
+----------+
```

This behavior is undesirable if you want only rows that contain both words. One way to do this is to rewrite the statement to look for each word separately and join the conditions with `AND`:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('David')
    -> AND MATCH(vtext) AGAINST('Goliath');
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

An easier way to require multiple words is with a Boolean mode search. To do this, precede each word in the search string with a `+` character and add `IN BOOLEAN MODE` after the string:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('+David +Goliath' IN BOOLEAN MODE);
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

Boolean mode searches also permit you to exclude words by preceding each one with a `-` character. The following queries select `kjv` rows containing the name David but not Goliath, and vice versa:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('+David -Goliath' IN BOOLEAN MODE);
+----------+
| COUNT(*) |
+----------+
|      892 |
+----------+
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('-David +Goliath' IN BOOLEAN MODE);
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

Another useful special character in Boolean searches is `*`; when appended to a search word, it acts as a wildcard operator. The following statement finds rows containing not only `whirl`, but also words such as `whirls`, `whirleth`, and `whirlwind`:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('whirl*' IN BOOLEAN MODE);
+----------+
| COUNT(*) |
+----------+
|       28 |
+----------+
```

For the complete list of Boolean full-text operators, see the *MySQL Reference Manual*.

5.15. Performing Full-Text Phrase Searches

Problem

You want to perform a full-text search for a phrase; that is, for words that occur adjacent to each other and in a specific order.

Solution

Use the full-text phrase-search capability.

Discussion

To find rows that contain a particular phrase, a simple full-text search doesn't work:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('still small voice');
+----------+
| COUNT(*) |
+----------+
|      548 |
+----------+
```

The query returns a result, but not the one you're looking for. A full-text search computes a relevance ranking based on the presence of each word individually, no matter where it occurs within the `vtext` column, and the ranking is nonzero as long as any of the words are present. Consequently, that kind of statement tends to find too many rows.

Instead, use full-text Boolean mode, which supports phrase searching. Enclose the phrase in double quotes within the search string:

```
mysql> SELECT COUNT(*) FROM kjv
    -> WHERE MATCH(vtext) AGAINST('"still small voice"' IN BOOLEAN MODE);
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
```

A phrase match succeeds if a column contains the same words as in the phrase, in the order specified.

CHAPTER 6

Working with Dates and Times

6.0. Introduction

MySQL has several data types for representing dates and times, and many functions for operating on them. MySQL stores dates and times in specific formats, and it's important to understand them to avoid surprises in results from manipulating temporal data. This chapter covers the following aspects of working with date and time values in MySQL:

Choosing a temporal data type
: MySQL provides several temporal data types to choose from when you create tables. Knowing their properties enables you to choose them appropriately.

Displaying dates and times
: MySQL displays temporal values using specific formats by default. You can produce other formats by using the appropriate functions.

Changing the client time zone
: The server interprets `TIMESTAMP` values in the client's current time zone, not its own. Clients in different time zones should set their zone so that the server can properly interpret `TIMESTAMP` values for them.

Determining the current date and time
: MySQL provides functions that return the date and time. These are useful for applications that must know these values or need to calculate other temporal values in relation to them.

Tracking row modification times
: The `TIMESTAMP` and `DATETIME` data types have special properties that enable you to record row-creation and last-modification times automatically.

Breaking dates and times into component values, creating dates and times from component values
: You can split date and time values when you need only a component, such as the month part of a date or the hour part of a time. Conversely, you can combine component values to synthesize dates and times.

Converting between dates or times and basic units
: Some temporal calculations such as date arithmetic operations are more easily performed using the number of days or seconds represented by a date or time value than by using the value itself. MySQL can perform conversions between date and time values and more basic units such as days or seconds.

Date and time arithmetic
: You can add or subtract temporal values to produce other temporal values or calculate intervals between values. Applications include age determination, relative date computation, and date shifting.

Selecting data based on temporal constraints
: The calculations discussed in the preceding sections to produce output values can also be used in `WHERE` clauses to specify how to select rows using temporal conditions.

This chapter covers several MySQL functions for operating on date and time values, but there are many others. To familiarize yourself with the full set, consult the *MySQL Reference Manual*. The variety of functions available to you means that it's often possible to perform a given temporal calculation more than one way. I sometimes illustrate alternative methods for achieving a given result, and many of the problems addressed in this chapter can be solved in ways other than shown here. I invite you to experiment to find other solutions. You may find a method that's more efficient or that you find more intuitive.

Scripts that implement recipes discussed in this chapter are located in the *dates* directory of the `recipes` source distribution. Scripts that create tables used here are located in the *tables* directory.

6.1. Choosing a Temporal Data Type

Problem

You need to store temporal data but aren't sure which is the most appropriate data type.

Solution

Choose the data type according to the characteristics of the information to be stored and how you need to use it.

Discussion

To choose a temporal data type, consider questions such as these:

- Do you need times only, dates only, or combined date and time values?
- What range of values do you require?
- Do you want automatic initialization of the column to the current date and time?

MySQL provides `DATE` and `TIME` data types for representing date and time values separately, and `DATETIME` and `TIMESTAMP` types for combined date-and-time values. These values have the following characteristics:

- `DATE` values have *`CCYY-MM-DD`* format, where *`CC`*, *`YY`*, *`MM`*, and *`DD`* represent the century, year within century, month, and day parts of the date. The supported range for `DATE` values is `1000-01-01` to `9999-12-31`.
- `TIME` values have *`hh:mm:ss`* format, where *`hh`*, *`mm`*, and *`ss`* are the hours, minutes, and seconds parts of the time. `TIME` values often can be thought of as time-of-day values, but MySQL actually treats them as elapsed time. Thus, they may be greater than `23:59:59` or even negative. (The actual range of a `TIME` column is `-838:59:59` to `838:59:59`.)
- `DATETIME` and `TIMESTAMP` are combined date-and-time values in *`CCYY-MM-DD hh:mm:ss`* format.

  The `DATETIME` and `TIMESTAMP` data types are similar in many respects, but watch out for these differences:

  — `DATETIME` has a supported range of `1000-01-01 00:00:00` to `9999-12-31 23:59:59`, whereas `TIMESTAMP` values are valid only from the year 1970 partially through 2038.

  — `TIMESTAMP` and `DATETIME` have special auto-initialization and auto-update properties (see Recipe 6.7), but for `DATETIME` they are not available before MySQL 5.6.5.

  — When a client inserts a `TIMESTAMP` value, the server converts it from the time zone associated with the client session to UTC and stores the UTC value. When the client retrieves a `TIMESTAMP` value, the server performs the reverse operation to convert the UTC value back to the client session time zone. A client in a time zone different from the server can configure its session so that this conversion is appropriate for its own time zone (see Recipe 6.4).
- Types that include a time part can have a fractional seconds part for subsecond resolution (see Recipe 6.2).

Many of the examples in this chapter draw on the following tables, which contain columns representing time, date, and date-and-time values. (The `time_val` table has two columns for use in time interval calculation examples.)

```
mysql> SELECT t1, t2 FROM time_val;
+----------+----------+
| t1       | t2       |
+----------+----------+
| 15:00:00 | 15:00:00 |
| 05:01:30 | 02:30:20 |
| 12:30:20 | 17:30:45 |
+----------+----------+
mysql> SELECT d FROM date_val;
+------------+
| d          |
+------------+
| 1864-02-28 |
| 1900-01-15 |
| 1999-12-31 |
| 2000-06-04 |
| 2017-03-16 |
+------------+
mysql> SELECT dt FROM datetime_val;
+---------------------+
| dt                  |
+---------------------+
| 1970-01-01 00:00:00 |
| 1999-12-31 09:00:00 |
| 2000-06-04 15:45:30 |
| 2017-03-16 12:30:15 |
+---------------------+
```

It is a good idea to create the `time_val`, `date_val`, and `datetime_val` tables right now before reading further. (Use the appropriate scripts in the *tables* directory of the `recipes` distribution.)

6.2. Using Fractional Seconds Support

As of MySQL 5.6.4, fractional seconds are supported for temporal types that include a time part: `DATETIME`, `TIME`, and `TIMESTAMP`. For applications that require subsecond resolution of time values, this enables you to specify fractional seconds precision down to the microsecond level.

The default is to have no fractional seconds part, so to include it for temporal types that support this capability, specify it explicitly in the column declaration: include (*`fsp`*) after the data type name in a column definition. *`fsp`* can be from 0 to 6 to indicate the number of fractional digits. 0 means "none" (resolution to seconds), 6 means resolution to microseconds. For example, to create a `TIME` column with two fractional digits (resolution to hundredths of seconds), use this syntax:

```
mycol TIME(2)
```

As an example, the 2014 Winter Olympics in Sochi are underway as I write. Scores for some Olympic events are measured as elapsed time, and events vary in the resolution used. The following table shows some representative events, their required time resolution for scores, and the `TIME` declaration appropriate for recording times of the competitors:

Event	Resolution	Data type
Biathlon	Tenths	`TIME(1)`
Downhill skiing	Hundredths	`TIME(2)`
Luge, skeleton	Thousandths	`TIME(3)`

Temporal functions that return current time or date-and-time values also support fractional seconds. The default without an argument is no fractional part. Otherwise, the argument specifies the desired resolution. Permitted values are 0 to 6, the same as when declaring temporal columns:

```
mysql> SELECT CURTIME(), CURTIME(2), CURTIME(6);
+-----------+-------------+-----------------+
| CURTIME() | CURTIME(2)  | CURTIME(6)      |
+-----------+-------------+-----------------+
| 18:07:03  | 18:07:03.24 | 18:07:03.244950 |
+-----------+-------------+-----------------+
```

6.3. Changing MySQL's Date Format

Problem

You want to change the ISO format that MySQL uses for representing date values.

Solution

You can't. However, you can rewrite non-ISO input values into ISO format when storing dates, and you can rewrite ISO values to other formats for display with the `DATE_FORMAT()` function.

Discussion

The *`CCYY-MM-DD`* format that MySQL uses for `DATE` values follows the ISO 8601 standard for representing dates. Because the year, month, and day parts have a fixed length and appear left to right in date strings, this format has the useful property that dates sort naturally into the proper temporal order. Recipes 7.5 and 8.12 discuss ordering and grouping techniques for date-based values.

ISO format, although common, is not used by all database systems, which can cause problems if you move data between different systems. Moreover, people commonly like to represent dates in other formats such as *MM/DD/YY* or *DD-MM-CCYY*. This too can be a source of trouble, due to mismatches between human expectations of how dates should look and how MySQL actually represents them.

A question frequently asked by newcomers to MySQL is, "How do I tell MySQL to store dates in a specific format such as *MM/DD/CCYY*?" That's the wrong question. Instead, ask, "If I have a date in a specific format, how can I store it in MySQL's supported format, and vice versa?" MySQL always stores dates in ISO format, a fact with implications both for data entry (input) and for displaying query results (output):

- For data-entry purposes, to store values that are not in ISO format, you normally must rewrite them first. If you don't want to rewrite them, you can store them as strings (for example, in a `CHAR` column). But then you can't operate on them as dates.

  Chapter 11 covers the topic of date rewriting for data entry, and Chapter 12 discusses checking dates to verify that they're valid. In some cases, if your values are close to ISO format, rewriting may not be necessary. For example, MySQL interprets the string values `87-1-7` and `1987-1-7` and the numbers `870107` and `19870107` as the date `1987-01-07` when storing them into a `DATE` column.
- For display purposes, you can rewrite dates to non-ISO formats. The `DATE_FORMAT()` function provides a lot of flexibility for changing date values into other formats (see later in this section). You can also use functions such as `YEAR()` to extract parts of dates for display (see Recipe 6.8). For additional discussion, see Recipe 12.14.

One way to rewrite non-ISO values for date entry is to use the `STR_TO_DATE()` function, which takes a string representing a temporal value and a format string that specifies the "syntax" of the value. Within the formatting string, use special sequences of the form `%`*c*, where *c* specifies which part of the date to expect. For example, `%Y`, `%M`, and `%d` signify the four-digit year, the month name, and the two-digit day of the month. To insert the value `May 13, 2007` into a `DATE` column, do this:

```
mysql> INSERT INTO t (d) VALUES(STR_TO_DATE('May 13, 2007','%M %d, %Y'));
mysql> SELECT d FROM t;
+------------+
| d          |
+------------+
| 2007-05-13 |
+------------+
```

For date display, MySQL uses ISO format (*CCYY-MM-DD*) unless you tell it otherwise. To display dates or times in other formats, use the `DATE_FORMAT()` or `TIME_FORMAT()` function to rewrite them. If you require a more specialized format those functions cannot provide, write a stored function.

The DATE_FORMAT() function takes two arguments: a DATE, DATETIME, or TIMESTAMP value, and a string describing how to display the value. The format string uses the same kind of specifiers as STR_TO_DATE(). The following statement shows the values in the date_val table, both as MySQL displays them by default and as reformatted with DATE_FORMAT():

```
mysql> SELECT d, DATE_FORMAT(d,'%M %d, %Y') FROM date_val;
+------------+--------------------------+
| d          | DATE_FORMAT(d,'%M %d, %Y') |
+------------+--------------------------+
| 1864-02-28 | February 28, 1864        |
| 1900-01-15 | January 15, 1900         |
| 1999-12-31 | December 31, 1999        |
| 2000-06-04 | June 04, 2000            |
| 2017-03-16 | March 16, 2017           |
+------------+--------------------------+
```

Because DATE_FORMAT() produces long column headings, it's often useful to provide an alias (see Recipe 3.2) to make a heading more concise or meaningful:

```
mysql> SELECT d, DATE_FORMAT(d,'%M %d, %Y') AS date FROM date_val;
+------------+------------------+
| d          | date             |
+------------+------------------+
| 1864-02-28 | February 28, 1864 |
| 1900-01-15 | January 15, 1900  |
| 1999-12-31 | December 31, 1999 |
| 2000-06-04 | June 04, 2000     |
| 2017-03-16 | March 16, 2017    |
+------------+------------------+
```

The *MySQL Reference Manual* provides a complete list of format sequences to use with DATE_FORMAT(), TIME_FORMAT(), and STR_TO_DATE(). The following table shows some of them:

Sequence	Meaning
%Y	Four-digit year
%y	Two-digit year
%M	Complete month name
%b	Month name, initial three letters
%m	Two-digit month of year (01..12)
%c	Month of year (1..12)
%d	Two-digit day of month (01..31)
%e	Day of month (1..31)
%W	Weekday name (Sunday..Saturday)
%r	12-hour time with AM or PM suffix
%T	24-hour time

Sequence	Meaning
%H	Two-digit hour
%i	Two-digit minute
%s	Two-digit second
%%	Literal %

The time-related format sequences shown in the table are useful only when you pass DATE_FORMAT() a value that has both date and time parts (a DATETIME or TIMESTAMP). The following statement displays DATETIME values from the datetime_val table using formats that include the time of day:

```
mysql> SELECT dt,
    -> DATE_FORMAT(dt,'%c/%e/%y %r') AS format1,
    -> DATE_FORMAT(dt,'%M %e, %Y %T') AS format2
    -> FROM datetime_val;
+---------------------+----------------------+---------------------------+
| dt                  | format1              | format2                   |
+---------------------+----------------------+---------------------------+
| 1970-01-01 00:00:00 | 1/1/70 12:00:00 AM   | January 1, 1970 00:00:00  |
| 1999-12-31 09:00:00 | 12/31/99 09:00:00 AM | December 31, 1999 09:00:00 |
| 2000-06-04 15:45:30 | 6/4/00 03:45:30 PM   | June 4, 2000 15:45:30     |
| 2017-03-16 12:30:15 | 3/16/17 12:30:15 PM  | March 16, 2017 12:30:15   |
+---------------------+----------------------+---------------------------+
```

TIME_FORMAT() is similar to DATE_FORMAT(). It works with TIME, DATETIME, or TIMESTAMP values, but understands only time-related specifiers in the format string:

```
mysql> SELECT dt,
    -> TIME_FORMAT(dt, '%r') AS '12-hour time',
    -> TIME_FORMAT(dt, '%T') AS '24-hour time'
    -> FROM datetime_val;
+---------------------+--------------+--------------+
| dt                  | 12-hour time | 24-hour time |
+---------------------+--------------+--------------+
| 1970-01-01 00:00:00 | 12:00:00 AM  | 00:00:00     |
| 1999-12-31 09:00:00 | 09:00:00 AM  | 09:00:00     |
| 2000-06-04 15:45:30 | 03:45:30 PM  | 15:45:30     |
| 2017-03-16 12:30:15 | 12:30:15 PM  | 12:30:15     |
+---------------------+--------------+--------------+
```

If DATE_FORMAT() or TIME_FORMAT() cannot produce the results that you want, write a stored function that does. Suppose that you want to convert 24-hour TIME values to 12-hour format but with a suffix of a.m. or p.m. rather than AM or PM. The following function accomplishes that task. It uses TIME_FORMAT() to do most of the work, then strips the suffix supplied by %r and replaces it with the desired suffix:

```
CREATE FUNCTION time_ampm (t TIME)
RETURNS VARCHAR(13) # mm:dd:ss {a.m.|p.m.} format
DETERMINISTIC
```

```
    RETURN CONCAT(LEFT(TIME_FORMAT(t, '%r'), 9),
                  IF(TIME_TO_SEC(t) < 12*60*60, 'a.m.', 'p.m.'));
```

Use the function like this:

```
mysql> SELECT t1, time_ampm(t1) FROM time_val;
+----------+---------------+
| t1       | time_ampm(t1) |
+----------+---------------+
| 15:00:00 | 03:00:00 p.m. |
| 05:01:30 | 05:01:30 a.m. |
| 12:30:20 | 12:30:20 p.m. |
+----------+---------------+
```

For more information about writing stored functions, see Chapter 9.

6.4. Setting the Client Time Zone

Problem

You have a client application that connects from a time zone different from the server. Consequently, when it stores `TIMESTAMP` values, they don't have the correct UTC values.

Solution

The client should set the `time_zone` system variable after connecting to the server.

Discussion

Time zone settings have an important effect on `TIMESTAMP` values:

- When the MySQL server starts, it examines its operating environment to determine its time zone. (To use a different value, start the server with the `--default-time-zone` option.)
- For each client that connects, the server interprets `TIMESTAMP` values with respect to the time zone associated with the client session. When a client inserts a `TIMESTAMP` value, the server converts it from the client time zone to UTC and stores the UTC value. (Internally, the server stores a `TIMESTAMP` value as the number of seconds since `1970-01-01 00:00:00` UTC.) When the client retrieves a `TIMESTAMP` value, the server performs the reverse operation to convert the UTC value back to the client time zone.
- The default session time zone for each client when it connects is the server time zone. If all clients are in the same time zone as the server, nothing special need be done for proper `TIMESTAMP` time zone conversion to occur. But if a client is in a time

zone different from the server and it inserts TIMESTAMP values without making the proper time zone correction, the UTC values won't be correct.

Suppose that the server and client C1 are in the same time zone, and client C1 issues these statements:

```
mysql> CREATE TABLE t (ts TIMESTAMP);
mysql> INSERT INTO t (ts) VALUES('2014-06-01 12:30:00');
mysql> SELECT ts FROM t;
+---------------------+
| ts                  |
+---------------------+
| 2014-06-01 12:30:00 |
+---------------------+
```

Here, client C1 sees the same value that it stored. A different client, C2, will also see the same value if it retrieves it, but if client C2 is in a different time zone, that value isn't correct for its zone. Conversely, if client C2 stores a value, that value when returned by client C1 won't be correct for the client C1 time zone.

To deal with this problem so that TIMESTAMP conversions use the proper time zone, a client should set its time zone explicitly by setting the session value of the time_zone system variable. Suppose that the server has a global time zone of six hours ahead of UTC. Each client initially is assigned that same value as its session time zone:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| +06:00             | +06:00              |
+--------------------+---------------------+
```

When client C2 connects, it sees the same TIMESTAMP value as client C1:

```
mysql> SELECT ts FROM t;
+---------------------+
| ts                  |
+---------------------+
| 2014-06-01 12:30:00 |
+---------------------+
```

But that value is incorrect if client C2 is only four hours ahead of UTC. C2 should set its time zone after connecting so that retrieved TIMESTAMP values are properly adjusted for its own session:

```
mysql> SET SESSION time_zone = '+04:00';
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| +06:00             | +04:00              |
+--------------------+---------------------+
```

```
mysql> SELECT ts FROM t;
+---------------------+
| ts                  |
+---------------------+
| 2014-06-01 10:30:00 |
+---------------------+
```

The client time zone also affects the values displayed from functions that return the current date and time (see Recipe 6.6).

See Also

To convert individual date-and-time values from one time zone to another, use the `CONVERT_TZ()` function (see Recipe 6.5).

6.5. Shifting Temporal Values Between Time Zones

Problem

You have a date-and-time value, but need to know what it would be in a different time zone. For example, you're having a teleconference with people in different parts of the world and you must tell them the meeting time in their local time zones.

Solution

Use the `CONVERT_TZ()` function.

Discussion

The `CONVERT_TZ()` function converts temporal values between time zones. It takes three arguments: a date-and-time value and two time zone indicators. The function interprets the date-and-time value as a value in the first time zone and returns the value shifted into the second time zone.

Suppose that I live in Chicago, Illinois in the US, and that I have a meeting with people in several other parts of the world. The following table shows the location of each meeting participant and the time zone name for each:

Location	Time zone name
Chicago, Illinois, US	`US/Central`
Berlin, Germany	`Europe/Berlin`
London, United Kingdom	`Europe/London`
Edmonton, Alberta, Canada	`America/Edmonton`
Brisbane, Australia	`Australia/Brisbane`

If the meeting is to take place at 9 AM local time for me on November 23, 2014, what time will that be for the other participants? The following statement uses `CONVERT_TZ()` to calculate the local times for each time zone:

```
mysql> SET @dt = '2014-11-23 09:00:00';
mysql> SELECT @dt AS Chicago,
    -> CONVERT_TZ(@dt,'US/Central','Europe/Berlin') AS Berlin,
    -> CONVERT_TZ(@dt,'US/Central','Europe/London') AS London,
    -> CONVERT_TZ(@dt,'US/Central','America/Edmonton') AS Edmonton,
    -> CONVERT_TZ(@dt,'US/Central','Australia/Brisbane') AS Brisbane\G
*************************** 1. row ***************************
 Chicago: 2014-11-23 09:00:00
  Berlin: 2014-11-23 16:00:00
  London: 2014-11-23 15:00:00
Edmonton: 2014-11-23 08:00:00
Brisbane: 2014-11-24 01:00:00
```

Let's hope the Brisbane participant doesn't mind being up after midnight.

The preceding example uses time zone names, so it requires that you have the time zone tables in the `mysql` database initialized with support for named time zones. (See the *MySQL Reference Manual* for information about setting up the time zone tables.) If you can't use named time zones, specify them in terms of their numeric relationship to UTC. (This can be a little trickier; you might need to account for daylight saving time.) The corresponding statement with numeric time zones looks like this:

```
mysql> SELECT @dt AS Chicago,
    -> CONVERT_TZ(@dt,'-06:00','+01:00') AS Berlin,
    -> CONVERT_TZ(@dt,'-06:00','+00:00') AS London,
    -> CONVERT_TZ(@dt,'-06:00','-07:00') AS Edmonton,
    -> CONVERT_TZ(@dt,'-06:00','+10:00') AS Brisbane\G
*************************** 1. row ***************************
 Chicago: 2014-11-23 09:00:00
  Berlin: 2014-11-23 16:00:00
  London: 2014-11-23 15:00:00
Edmonton: 2014-11-23 08:00:00
Brisbane: 2014-11-24 01:00:00
```

6.6. Determining the Current Date or Time

Problem

What's today's date? What time is it?

Solution

Use the `CURDATE()`, `CURTIME()`, or `NOW()` functions to obtain values expressed in the client session time zone. Use `UTC_DATE()`, `UTC_TIME()`, or `UTC_TIMESTAMP()` for values in UTC time.

Discussion

Some applications must know the current date or time, such as those that write time-stamped log records. This kind of information is also useful for date calculations performed in relation to the current date, such as finding the first (or last) day of the month, or determining the date for Wednesday of next week.

The `CURDATE()` and `CURTIME()` functions return the current date and time separately, and `NOW()` returns both as a combined date-and-time value:

```
mysql> SELECT CURDATE(), CURTIME(), NOW();
+------------+-----------+---------------------+
| CURDATE()  | CURTIME() | NOW()               |
+------------+-----------+---------------------+
| 2014-02-20 | 18:06:45  | 2014-02-20 18:06:45 |
+------------+-----------+---------------------+
```

`CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` are synonyms for `CURDATE()`, `CURTIME()`, and `NOW()`, respectively.

The preceding functions return values in the client session time zone (see Recipe 6.4). For values in UTC time, use the `UTC_DATE()`, `UTC_TIME()`, or `UTC_TIMESTAMP()` functions instead.

To determine the current date and time for an arbitrary time zone, pass the value of the appropriate UTC function to `CONVERT_TZ()` (see Recipe 6.5).

To obtain subparts of these values, such as the current day of the month or current hour of the day, use the decomposition techniques discussed in Recipe 6.8.

6.7. Using TIMESTAMP or DATETIME to Track Row-Modification Times

Problem

You want to record row-creation time or last modification time automatically.

Solution

Use the auto-initialization and auto-update properties of the `TIMESTAMP` and `DATETIME` data types.

Discussion

MySQL supports `TIMESTAMP` and `DATETIME` data types that store date-and-time values. Recipe 6.1 covers the range of values for these types. This section focuses on special

column attributes that enable you to track row-creation and -update times automatically:

- A `TIMESTAMP` or `DATETIME` column declared with the `DEFAULT CURRENT_TIMESTAMP` attribute initializes automatically for new rows. Simply omit the column from `INSERT` statements and MySQL sets it to the row-creation time.
- A `TIMESTAMP` or `DATETIME` column declared with the `ON UPDATE CURRENT_TIMESTAMP` attribute automatically updates to the current date and time when you change any other column in the row from its current value.

These special properties make the `TIMESTAMP` and `DATETIME` data types particularly suited for applications that require recording the times at which rows are inserted or updated. The following discussion shows how to take advantage of these properties using `TIMESTAMP` columns. With some differences to be noted later, the discussion also applies to `DATETIME` columns.

This section assumes that you have MySQL 5.6.5 or later. For older versions, automatic initialization and update properties apply only to `TIMESTAMP` (not `DATETIME`), and to at most a single `TIMESTAMP` in a table.

Our example table looks like this:

```
CREATE TABLE tsdemo
(
  val INT,
  ts_both   TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  ts_create TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  ts_update TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

The `TIMESTAMP` columns have these properties:

- `ts_both` auto-initializes and auto-updates. This is useful for tracking the time of any change to a row, for both inserts and updates.
- `ts_create` auto-initializes only. This is useful when you want a column to be set to the time at which a row is created, but remain constant thereafter.
- `ts_update` auto-updates only. It is set to the column default (or value you specify explicitly) at row-creation time and it auto-updates for changes to the row thereafter. The use cases for this are more limited—for example, to track row-creation and last-modification times separately (using `ts_update` in conjunction with `ts_create`), rather than together in a single column like `ts_both`.

To see how the table works, insert some rows into the table (a few seconds apart so the timestamps differ), then select its contents:

```
mysql> INSERT INTO tsdemo (val) VALUES(5);
mysql> INSERT INTO tsdemo (val,ts_both,ts_create,ts_update)
    -> VALUES(10,NULL,NULL,NULL);
mysql> SELECT * FROM tsdemo;
+------+---------------------+---------------------+---------------------+
| val  | ts_both             | ts_create           | ts_update           |
+------+---------------------+---------------------+---------------------+
|    5 | 2014-02-20 18:06:45 | 2014-02-20 18:06:45 | 0000-00-00 00:00:00 |
|   10 | 2014-02-20 18:06:50 | 2014-02-20 18:06:50 | 2014-02-20 18:06:50 |
+------+---------------------+---------------------+---------------------+
```

The first `INSERT` statement shows that you can set the auto-initialize columns to the current date and time by omitting them from the `INSERT` statement entirely. The second shows that you can set a `TIMESTAMP` column to the current date and time by setting it explicitly to `NULL`, even one that does not auto-initialize. This `NULL`-assignment behavior is not specific to `INSERT` statements; it works for `UPDATE` as well. You can disable this special handling of `NULL` assignments, as we'll cover later in this section.

To see auto-updating in action, issue a statement that changes one row's `val` column and check its effect on the table's contents. The result shows that the auto-update columns are updated (in the modified row only):

```
mysql> UPDATE tsdemo SET val = 11 WHERE val = 10;
mysql> SELECT * FROM tsdemo;
+------+---------------------+---------------------+---------------------+
| val  | ts_both             | ts_create           | ts_update           |
+------+---------------------+---------------------+---------------------+
|    5 | 2014-02-20 18:06:45 | 2014-02-20 18:06:45 | 0000-00-00 00:00:00 |
|   11 | 2014-02-20 18:06:55 | 2014-02-20 18:06:50 | 2014-02-20 18:06:55 |
+------+---------------------+---------------------+---------------------+
```

If you modify multiple rows, updates occur for the auto-update columns in each row:

```
mysql> UPDATE tsdemo SET val = val + 1;
mysql> SELECT * FROM tsdemo;
+------+---------------------+---------------------+---------------------+
| val  | ts_both             | ts_create           | ts_update           |
+------+---------------------+---------------------+---------------------+
|    6 | 2014-02-20 18:07:01 | 2014-02-20 18:06:45 | 2014-02-20 18:07:01 |
|   12 | 2014-02-20 18:07:01 | 2014-02-20 18:06:50 | 2014-02-20 18:07:01 |
+------+---------------------+---------------------+---------------------+
```

An `UPDATE` statement that doesn't actually change any value in a row doesn't modify auto-update columns. To see this, set every row's `val` column to its current value, then review the table contents to see that auto-update columns retain their values:

```
mysql> UPDATE tsdemo SET val = val;
mysql> SELECT * FROM tsdemo;
```

```
+------+---------------------+---------------------+---------------------+
| val  | ts_both             | ts_create           | ts_update           |
+------+---------------------+---------------------+---------------------+
|    6 | 2014-02-20 18:07:01 | 2014-02-20 18:06:45 | 2014-02-20 18:07:01 |
|   12 | 2014-02-20 18:07:01 | 2014-02-20 18:06:50 | 2014-02-20 18:07:01 |
+------+---------------------+---------------------+---------------------+
```

As stated previously, automatic `TIMESTAMP` properties also apply to `DATETIME`, with some differences:

- For the first `TIMESTAMP` column in a table, if neither of the `DEFAULT` or `ON UPDATE` attributes are specified, the column is implicitly defined with both. For `DATETIME`, automatic properties never apply implicitly; only those specified explicitly. (To suppress implicit attribute definition for `TIMESTAMP` columns, enable the `explicit_defaults_for_timestamp` system variable.)
- You can set a `TIMESTAMP` column to the current date and time at any time by setting it to `NULL`, unless it has specifically been defined to permit `NULL` values. Assigning `NULL` to a `DATETIME` column never sets it to the current date and time.

  To prevent a `TIMESTAMP` column from being set to the current date and time when assigned a `NULL` value, include the `NULL` attribute in the column definition. Then assigning `NULL` to the column stores `NULL`.

To determine for any given `TIMESTAMP` column what happens when `NULL` is assigned to it, use `SHOW CREATE TABLE` to see the column definition. If the definition includes the `NULL` attribute, assigning `NULL` stores `NULL`. If the definition includes the `NOT NULL` attribute, you can specify `NULL` as the value to be assigned, but you cannot *store* `NULL` because MySQL stores the current date and time instead.

See Also

To simulate `TIMESTAMP` auto-initialization and auto-update properties for other temporal types, you can use triggers (see Recipe 9.6). This technique applies to `DATE` and `TIME`, and can also be useful for `DATETIME` before MySQL 5.6.5 (when automatic properties for that type were introduced).

6.8. Extracting Parts of Dates or Times

Problem

You want to obtain just a part of a date or a time.

Solution

Invoke a function specifically intended for extracting part of a temporal value, such as `MONTH()` or `MINUTE()`. This is usually the fastest method for component extraction if you need only a single component of a value. Alternatively, use a formatting function such as `DATE_FORMAT()` or `TIME_FORMAT()` with a format string that includes a specifier for the part of the value you want to obtain.

Discussion

The following discussion shows different ways to extract parts of temporal values.

Decomposing dates or times using component-extraction functions

MySQL includes many functions for extracting date and time subpart extraction. For example, `DATE()` and `TIME()` extract the date and time components of temporal values:

```
mysql> SELECT dt, DATE(dt), TIME(dt) FROM datetime_val;
+---------------------+------------+----------+
| dt                  | DATE(dt)   | TIME(dt) |
+---------------------+------------+----------+
| 1970-01-01 00:00:00 | 1970-01-01 | 00:00:00 |
| 1999-12-31 09:00:00 | 1999-12-31 | 09:00:00 |
| 2000-06-04 15:45:30 | 2000-06-04 | 15:45:30 |
| 2017-03-16 12:30:15 | 2017-03-16 | 12:30:15 |
+---------------------+------------+----------+
```

The following table shows some several component-extraction functions; consult the *MySQL Reference Manual* for a complete list. The date-related functions work with `DATE`, `DATETIME`, or `TIMESTAMP` values. The time-related functions work with `TIME`, `DATETIME`, or `TIMESTAMP` values:

Function	Return value
`YEAR()`	Year of date
`MONTH()`	Month number (1..12)
`MONTHNAME()`	Month name (January..December)
`DAYOFMONTH()`	Day of month (1..31)
`DAYNAME()`	Day name (Sunday..Saturday)
`DAYOFWEEK()`	Day of week (1..7 for Sunday..Saturday)
`WEEKDAY()`	Day of week (0..6 for Monday..Sunday)
`DAYOFYEAR()`	Day of year (1..366)
`HOUR()`	Hour of time (0..23)
`MINUTE()`	Minute of time (0..59)
`SECOND()`	Second of time (0..59)
`EXTRACT()`	Varies

Here's an example:

```
mysql> SELECT dt, YEAR(dt), DAYOFMONTH(dt), HOUR(dt), SECOND(dt)
    -> FROM datetime_val;
+---------------------+----------+---------------+----------+------------+
| dt                  | YEAR(dt) | DAYOFMONTH(dt) | HOUR(dt) | SECOND(dt) |
+---------------------+----------+---------------+----------+------------+
| 1970-01-01 00:00:00 |     1970 |             1 |        0 |          0 |
| 1999-12-31 09:00:00 |     1999 |            31 |        9 |          0 |
| 2000-06-04 15:45:30 |     2000 |             4 |       15 |         30 |
| 2017-03-16 12:30:15 |     2017 |            16 |       12 |         15 |
+---------------------+----------+---------------+----------+------------+
```

Functions such as `YEAR()` or `DAYOFMONTH()` extract values that have an obvious correspondence to a substring of the temporal value to which you apply them. Other component-extraction functions provide access to values that have no such correspondence. One is the day-of-year value:

```
mysql> SELECT d, DAYOFYEAR(d) FROM date_val;
+------------+--------------+
| d          | DAYOFYEAR(d) |
+------------+--------------+
| 1864-02-28 |           59 |
| 1900-01-15 |           15 |
| 1999-12-31 |          365 |
| 2000-06-04 |          156 |
| 2017-03-16 |           75 |
+------------+--------------+
```

Another is the day of the week, which is available by name or number:

- `DAYNAME()` returns the complete day name. There is no function for returning the three-character name abbreviation, but you can get it easily by passing the full name to `LEFT()`:

  ```
  mysql> SELECT d, DAYNAME(d), LEFT(DAYNAME(d),3) FROM date_val;
  +------------+------------+--------------------+
  | d          | DAYNAME(d) | LEFT(DAYNAME(d),3) |
  +------------+------------+--------------------+
  | 1864-02-28 | Sunday     | Sun                |
  | 1900-01-15 | Monday     | Mon                |
  | 1999-12-31 | Friday     | Fri                |
  | 2000-06-04 | Sunday     | Sun                |
  | 2017-03-16 | Thursday   | Thu                |
  +------------+------------+--------------------+
  ```

- To get the day of the week as a number, use `DAYOFWEEK()` or `WEEKDAY()`, but pay attention to the range of values each function returns. `DAYOFWEEK()` returns values from 1 to 7, corresponding to Sunday through Saturday. `WEEKDAY()` returns values from 0 to 6, corresponding to Monday through Sunday:

  ```
  mysql> SELECT d, DAYNAME(d), DAYOFWEEK(d), WEEKDAY(d) FROM date_val;
  +------------+------------+--------------+------------+
  ```

```
| d          | DAYNAME(d) | DAYOFWEEK(d) | WEEKDAY(d) |
+------------+------------+--------------+------------+
| 1864-02-28 | Sunday     |            1 |          6 |
| 1900-01-15 | Monday     |            2 |          0 |
| 1999-12-31 | Friday     |            6 |          4 |
| 2000-06-04 | Sunday     |            1 |          6 |
| 2017-03-16 | Thursday   |            5 |          3 |
+------------+------------+--------------+------------+
```

`EXTRACT()` is another function for obtaining individual parts of temporal values:

```
mysql> SELECT dt, EXTRACT(DAY FROM dt), EXTRACT(HOUR FROM dt)
    -> FROM datetime_val;
+---------------------+----------------------+----------------------+
| dt                  | EXTRACT(DAY FROM dt) | EXTRACT(HOUR FROM dt) |
+---------------------+----------------------+----------------------+
| 1970-01-01 00:00:00 |                    1 |                    0 |
| 1999-12-31 09:00:00 |                   31 |                    9 |
| 2000-06-04 15:45:30 |                    4 |                   15 |
| 2017-03-16 12:30:15 |                   16 |                   12 |
+---------------------+----------------------+----------------------+
```

The keyword indicating what to extract from the value should be a unit specifier such as `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, or `SECOND`. Unit specifiers are singular, not plural. (Check the *MySQL Reference Manual* for the full list.)

Obtaining the Current Year, Month, Day, Hour, Minute, or Second

To obtain the current year, month, day, or day of week, apply the extraction functions shown in this recipe to `CURDATE()` or `NOW()`:

```
mysql> SELECT CURDATE(), YEAR(CURDATE()) AS year,
    -> MONTH(CURDATE()) AS month, MONTHNAME(CURDATE()) AS monthname,
    -> DAYOFMONTH(CURDATE()) AS day, DAYNAME(CURDATE()) AS dayname;
+------------+------+-------+-----------+------+----------+
| CURDATE()  | year | month | monthname | day  | dayname  |
+------------+------+-------+-----------+------+----------+
| 2014-02-20 | 2014 |     2 | February  |   20 | Thursday |
+------------+------+-------+-----------+------+----------+
```

Similarly, to obtain the current hour, minute, or second, pass `CURTIME()` or `NOW()` to a time-component function:

```
mysql> SELECT NOW(), HOUR(NOW()) AS hour,
    -> MINUTE(NOW()) AS minute, SECOND(NOW()) AS second;
+---------------------+------+--------+--------+
| NOW()               | hour | minute | second |
+---------------------+------+--------+--------+
| 2014-02-20 18:07:03 |   18 |      7 |      3 |
+---------------------+------+--------+--------+
```

Decomposing dates or times using formatting functions

The `DATE_FORMAT()` and `TIME_FORMAT()` functions reformat date and time values. By specifying appropriate format strings, you can extract individual parts of temporal values:

```
mysql> SELECT dt,
    -> DATE_FORMAT(dt,'%Y') AS year,
    -> DATE_FORMAT(dt,'%d') AS day,
    -> TIME_FORMAT(dt,'%H') AS hour,
    -> TIME_FORMAT(dt,'%s') AS second
    -> FROM datetime_val;
+---------------------+------+------+------+--------+
| dt                  | year | day  | hour | second |
+---------------------+------+------+------+--------+
| 1970-01-01 00:00:00 | 1970 | 01   | 00   | 00     |
| 1999-12-31 09:00:00 | 1999 | 31   | 09   | 00     |
| 2000-06-04 15:45:30 | 2000 | 04   | 15   | 30     |
| 2017-03-16 12:30:15 | 2017 | 16   | 12   | 15     |
+---------------------+------+------+------+--------+
```

Formatting functions are advantageous when you want to extract more than one part of a value, or display extracted values in a format different from the default. For example, to extract the entire date or time from `DATETIME` values, do this:

```
mysql> SELECT dt,
    -> DATE_FORMAT(dt,'%Y-%m-%d') AS 'date part',
    -> TIME_FORMAT(dt,'%T') AS 'time part'
    -> FROM datetime_val;
+---------------------+------------+-----------+
| dt                  | date part  | time part |
+---------------------+------------+-----------+
| 1970-01-01 00:00:00 | 1970-01-01 | 00:00:00  |
| 1999-12-31 09:00:00 | 1999-12-31 | 09:00:00  |
| 2000-06-04 15:45:30 | 2000-06-04 | 15:45:30  |
| 2017-03-16 12:30:15 | 2017-03-16 | 12:30:15  |
+---------------------+------------+-----------+
```

To present a date in other than *`CCYY-MM-DD`* format or a time without the seconds part, do this:

```
mysql> SELECT dt,
    -> DATE_FORMAT(dt,'%M %e, %Y') AS 'descriptive date',
    -> TIME_FORMAT(dt,'%H:%i') AS 'hours/minutes'
    -> FROM datetime_val;
+---------------------+------------------+---------------+
| dt                  | descriptive date | hours/minutes |
+---------------------+------------------+---------------+
| 1970-01-01 00:00:00 | January 1, 1970  | 00:00         |
| 1999-12-31 09:00:00 | December 31, 1999 | 09:00        |
| 2000-06-04 15:45:30 | June 4, 2000     | 15:45         |
| 2017-03-16 12:30:15 | March 16, 2017   | 12:30         |
+---------------------+------------------+---------------+
```

6.9. Synthesizing Dates or Times from Component Values

Problem

You want to combine the parts of a date or time to produce a complete date or time value. Or you want to replace parts of a date to produce another date.

Solution

You have several options:

- Use `MAKETIME()` to construct a `TIME` value from hour, minute, and second parts.
- Use `DATE_FORMAT()` or `TIME_FORMAT()` to combine parts of the existing value with parts you want to replace.
- Pull out the parts that you need with component-extraction functions and recombine the parts with `CONCAT()`.

Discussion

The reverse of splitting a date or time value into components is synthesizing a temporal value from its constituent parts. Techniques for date and time synthesis include using composition functions, formatting functions, and string concatenation.

The `MAKETIME()` function takes component hour, minute, and second values as arguments and combines them to produce a time:

```
mysql> SELECT MAKETIME(10,30,58), MAKETIME(-5,0,11);
+--------------------+-------------------+
| MAKETIME(10,30,58) | MAKETIME(-5,0,11) |
+--------------------+-------------------+
| 10:30:58           | -05:00:11         |
+--------------------+-------------------+
```

Date synthesis often is performed beginning with a given date, then keeping parts that you want to use and replacing the rest. For example, to produce the first day of the month in which a date falls, use `DATE_FORMAT()` to extract the year and month parts from the date, combining them with a day part of `01`:

```
mysql> SELECT d, DATE_FORMAT(d,'%Y-%m-01') FROM date_val;
+------------+--------------------------+
| d          | DATE_FORMAT(d,'%Y-%m-01') |
+------------+--------------------------+
| 1864-02-28 | 1864-02-01               |
| 1900-01-15 | 1900-01-01               |
| 1999-12-31 | 1999-12-01               |
| 2000-06-04 | 2000-06-01               |
```

```
| 2017-03-16 | 2017-03-01               |
+------------+--------------------------+
```

`TIME_FORMAT()` can be used similarly. The following example produces time values that have the seconds part set to `00`:

```
mysql> SELECT t1, TIME_FORMAT(t1,'%H:%i:00') FROM time_val;
+----------+--------------------------+
| t1       | TIME_FORMAT(t1,'%H:%i:00') |
+----------+--------------------------+
| 15:00:00 | 15:00:00                 |
| 05:01:30 | 05:01:00                 |
| 12:30:20 | 12:30:00                 |
+----------+--------------------------+
```

Another way to construct temporal values is to use date-part extraction functions in conjunction with `CONCAT()`. However, this method often is messier than the `DATE_FORMAT()` technique just discussed, and it sometimes yields slightly different results:

```
mysql> SELECT d, CONCAT(YEAR(d),'-',MONTH(d),'-01') FROM date_val;
+------------+-----------------------------------+
| d          | CONCAT(YEAR(d),'-',MONTH(d),'-01') |
+------------+-----------------------------------+
| 1864-02-28 | 1864-2-01                         |
| 1900-01-15 | 1900-1-01                         |
| 1999-12-31 | 1999-12-01                        |
| 2000-06-04 | 2000-6-01                         |
| 2017-03-16 | 2017-3-01                         |
+------------+-----------------------------------+
```

Note that the month values in some of these dates have only a single digit. To ensure that the month has two digits—as required for ISO format—use `LPAD()` to add a leading zero as necessary:

```
mysql> SELECT d, CONCAT(YEAR(d),'-',LPAD(MONTH(d),2,'0'),'-01')
    -> FROM date_val;
+------------+-------------------------------------------------+
| d          | CONCAT(YEAR(d),'-',LPAD(MONTH(d),2,'0'),'-01') |
+------------+-------------------------------------------------+
| 1864-02-28 | 1864-02-01                                      |
| 1900-01-15 | 1900-01-01                                      |
| 1999-12-31 | 1999-12-01                                      |
| 2000-06-04 | 2000-06-01                                      |
| 2017-03-16 | 2017-03-01                                      |
+------------+-------------------------------------------------+
```

Recipe 6.19 shows other ways to solve the problem of producing ISO dates from not-quite-ISO dates.

`TIME` values can be produced from hours, minutes, and seconds values using methods analogous to those for creating `DATE` values. For example, to change a `TIME` value so that

its seconds part is `00`, extract the hour and minute parts, and then recombine them with `CONCAT()`:

```
mysql> SELECT t1,
    -> CONCAT(LPAD(HOUR(t1),2,'0'),':',LPAD(MINUTE(t1),2,'0'),':00')
    ->   AS recombined
    -> FROM time_val;
+----------+------------+
| t1       | recombined |
+----------+------------+
| 15:00:00 | 15:00:00   |
| 05:01:30 | 05:01:00   |
| 12:30:20 | 12:30:00   |
+----------+------------+
```

To produce a combined date-and-time value from separate date and time values, simply concatenate them separated by a space:

```
mysql> SET @d = '2014-02-28', @t = '13:10:05';
mysql> SELECT @d, @t, CONCAT(@d,' ',@t);
+------------+----------+---------------------+
| @d         | @t       | CONCAT(@d,' ',@t)   |
+------------+----------+---------------------+
| 2014-02-28 | 13:10:05 | 2014-02-28 13:10:05 |
+------------+----------+---------------------+
```

6.10. Converting Between Temporal Values and Basic Units

Problem

You want to convert a temporal value such as a time or date to basic units such as seconds or days. This is often useful or necessary for performing temporal arithmetic operations (see Recipes 6.11 and 6.12).

Solution

The conversion method depends on the type of value to be converted:

- To convert between time values and seconds, use the `TIME_TO_SEC()` and `SEC_TO_TIME()` functions.
- To convert between date values and days, use the `TO_DAYS()` and `FROM_DAYS()` functions.
- To convert between date-and-time values and seconds, use the `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` functions.

Discussion

The following discussion shows how to convert several types of temporal values to basic units and vice versa.

Converting between times and seconds

`TIME` values are specialized representations of a simpler unit (seconds). To convert from one to the other, use the `TIME_TO_SEC()` and `SEC_TO_TIME()` functions.

`TIME_TO_SEC()` converts a `TIME` value to the equivalent number of seconds, and `SEC_TO_TIME()` does the opposite. The following statement demonstrates a simple conversion in both directions:

```
mysql> SELECT t1,
    -> TIME_TO_SEC(t1) AS 'TIME to seconds',
    -> SEC_TO_TIME(TIME_TO_SEC(t1)) AS 'TIME to seconds to TIME'
    -> FROM time_val;
+----------+-----------------+-------------------------+
| t1       | TIME to seconds | TIME to seconds to TIME |
+----------+-----------------+-------------------------+
| 15:00:00 |           54000 | 15:00:00                |
| 05:01:30 |           18090 | 05:01:30                |
| 12:30:20 |           45020 | 12:30:20                |
+----------+-----------------+-------------------------+
```

To express time values as minutes, hours, or days, perform the appropriate divisions:

```
mysql> SELECT t1,
    -> TIME_TO_SEC(t1) AS 'seconds',
    -> TIME_TO_SEC(t1)/60 AS 'minutes',
    -> TIME_TO_SEC(t1)/(60*60) AS 'hours',
    -> TIME_TO_SEC(t1)/(24*60*60) AS 'days'
    -> FROM time_val;
+----------+---------+----------+---------+--------+
| t1       | seconds | minutes  | hours   | days   |
+----------+---------+----------+---------+--------+
| 15:00:00 |   54000 | 900.0000 | 15.0000 | 0.6250 |
| 05:01:30 |   18090 | 301.5000 |  5.0250 | 0.2094 |
| 12:30:20 |   45020 | 750.3333 | 12.5056 | 0.5211 |
+----------+---------+----------+---------+--------+
```

Use `FLOOR()` on the division results if you prefer integer values that have no fractional part.

If you pass `TIME_TO_SEC()` a date-and-time value, it extracts the time part and discards the date. This provides another means of extracting times from `DATETIME` (or `TIMESTAMP`) values, in addition to those already discussed in Recipe 6.8:

```
mysql> SELECT dt,
    -> TIME_TO_SEC(dt) AS 'time part in seconds',
    -> SEC_TO_TIME(TIME_TO_SEC(dt)) AS 'time part as TIME'
```

```
    -> FROM datetime_val;
+---------------------+----------------------+------------------+
| dt                  | time part in seconds | time part as TIME |
+---------------------+----------------------+------------------+
| 1970-01-01 00:00:00 |                    0 | 00:00:00          |
| 1999-12-31 09:00:00 |                32400 | 09:00:00          |
| 2000-06-04 15:45:30 |                56730 | 15:45:30          |
| 2017-03-16 12:30:15 |                45015 | 12:30:15          |
+---------------------+----------------------+------------------+
```

Converting between dates and days

If you have a date but want a value in days, or vice versa, use the `TO_DAYS()` and `FROM_DAYS()` functions. Date-and-time values also can be converted to days if you can suffer loss of the time part.

`TO_DAYS()` converts a date to the corresponding number of days, and `FROM_DAYS()` does the opposite:

```
mysql> SELECT d,
    -> TO_DAYS(d) AS 'DATE to days',
    -> FROM_DAYS(TO_DAYS(d)) AS 'DATE to days to DATE'
    -> FROM date_val;
+------------+--------------+----------------------+
| d          | DATE to days | DATE to days to DATE |
+------------+--------------+----------------------+
| 1864-02-28 |       680870 | 1864-02-28           |
| 1900-01-15 |       693975 | 1900-01-15           |
| 1999-12-31 |       730484 | 1999-12-31           |
| 2000-06-04 |       730640 | 2000-06-04           |
| 2017-03-16 |       736769 | 2017-03-16           |
+------------+--------------+----------------------+
```

When using `TO_DAYS()`, it's best to stick to the advice of the *MySQL Reference Manual* and avoid `DATE` values that occur before the beginning of the Gregorian calendar (1582). Changes in the lengths of calendar years and months prior to that date make it difficult to speak meaningfully of what the value of "day 0" might be. This differs from `TIME_TO_SEC()`, where the correspondence between a `TIME` value and the resulting seconds value is obvious and has a meaningful reference point of 0 seconds.

If you pass `TO_DAYS()` a date-and-time value, it extracts the date part and discards the time. This provides another means of extracting dates from `DATETIME` (or `TIMESTAMP`) values, in addition to those already discussed in Recipe 6.8:

```
mysql> SELECT dt,
    -> TO_DAYS(dt) AS 'date part in days',
    -> FROM_DAYS(TO_DAYS(dt)) AS 'date part as DATE'
    -> FROM datetime_val;
+---------------------+------------------+------------------+
| dt                  | date part in days | date part as DATE |
+---------------------+------------------+------------------+
```

```
| 1970-01-01 00:00:00 |            719528 | 1970-01-01        |
| 1999-12-31 09:00:00 |            730484 | 1999-12-31        |
| 2000-06-04 15:45:30 |            730640 | 2000-06-04        |
| 2017-03-16 12:30:15 |            736769 | 2017-03-16        |
+---------------------+-------------------+-------------------+
```

Converting between date-and-time values and seconds

For `DATETIME` or `TIMESTAMP` values that lie within the range of the `TIMESTAMP` data type (from the beginning of 1970 partially through 2038), the `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` functions convert to and from the number of seconds elapsed since the beginning of 1970. The conversion to seconds offers higher precision for date-and-time values than a conversion to days, at the cost of a more limited range of values for which the conversion may be performed (`TIME_TO_SEC()` is unsuitable for this because it discards the date):

```
mysql> SELECT dt,
    -> UNIX_TIMESTAMP(dt) AS seconds,
    -> FROM_UNIXTIME(UNIX_TIMESTAMP(dt)) AS timestamp
    -> FROM datetime_val;
+---------------------+------------+---------------------+
| dt                  | seconds    | timestamp           |
+---------------------+------------+---------------------+
| 1970-01-01 00:00:00 |      21600 | 1970-01-01 00:00:00 |
| 1999-12-31 09:00:00 |  946652400 | 1999-12-31 09:00:00 |
| 2000-06-04 15:45:30 |  960151530 | 2000-06-04 15:45:30 |
| 2017-03-16 12:30:15 | 1489685415 | 2017-03-16 12:30:15 |
+---------------------+------------+---------------------+
```

There is a relationship between "UNIX" in the function names and the fact that the applicable range of values begins with 1970: the "Unix epoch" begins at `1970-01-01 00:00:00` UTC. The epoch is time zero, or the reference point for measuring time in Unix systems. That being so, you may find it curious that the preceding example shows a `UNIX_TIMESTAMP()` value of `21600` for the first value in the `datetime_val` table. Why isn't it `0`? The apparent discrepancy is due to the fact that the MySQL server interprets the `UNIX_TIMESTAMP()` argument as a value in the client's local time zone and converts it to UTC (see Recipe 6.4). My server is in the US Central time zone, six hours (21,600 seconds) west of UTC. Change the session time zone to `'+00:00'` for UTC time and run the query again to observe a different result:

```
mysql> set time_zone = '+00:00';
mysql> SELECT dt,
    -> UNIX_TIMESTAMP(dt) AS seconds,
    -> FROM_UNIXTIME(UNIX_TIMESTAMP(dt)) AS timestamp
    -> FROM datetime_val;
+---------------------+------------+---------------------+
| dt                  | seconds    | timestamp           |
+---------------------+------------+---------------------+
| 1970-01-01 00:00:00 |          0 | 1970-01-01 00:00:00 |
| 1999-12-31 09:00:00 |  946630800 | 1999-12-31 09:00:00 |
```

```
| 2000-06-04 15:45:30 |  960133530 | 2000-06-04 15:45:30 |
| 2017-03-16 12:30:15 | 1489667415 | 2017-03-16 12:30:15 |
+---------------------+------------+---------------------+
```

`UNIX_TIMESTAMP()` can convert `DATE` values to seconds, too. It treats such values as having an implicit time-of-day part of `00:00:00`:

```
mysql> SELECT
    -> CURDATE(),
    -> UNIX_TIMESTAMP(CURDATE()),
    -> FROM_UNIXTIME(UNIX_TIMESTAMP(CURDATE()))\G
*************************** 1. row ***************************
                           CURDATE(): 2014-02-20
              UNIX_TIMESTAMP(CURDATE()): 1392876000
FROM_UNIXTIME(UNIX_TIMESTAMP(CURDATE())): 2014-02-20 00:00:00
```

6.11. Calculating Intervals Between Dates or Times

Problem

You want to know how long it is between two dates or times; that is, the interval between them.

Solution

To calculate an interval, use one of the temporal-difference functions, or convert your values to basic units and take the difference. The permitted functions depend on the types of the values for which you want to know the interval.

Discussion

The following discussion shows several ways to perform interval calculations.

Calculating intervals with temporal-difference functions

To calculate an interval in days between two date values, use the `DATEDIFF()` function:

```
mysql> SET @d1 = '2010-01-01', @d2 = '2009-12-01';
mysql> SELECT DATEDIFF(@d1,@d2) AS 'd1 - d2', DATEDIFF(@d2,@d1) AS 'd2 - d1';
+---------+---------+
| d1 - d2 | d2 - d1 |
+---------+---------+
|      31 |     -31 |
+---------+---------+
```

`DATEDIFF()` also works with date-and-time values, but ignores the time part. This makes it suitable for producing day intervals for `DATE`, `DATETIME`, or `TIMESTAMP` values.

To calculate an interval between TIME values as another TIME value, use the TIMEDIFF() function:

```
mysql> SET @t1 = '12:00:00', @t2 = '16:30:00';
mysql> SELECT TIMEDIFF(@t1,@t2) AS 't1 - t2', TIMEDIFF(@t2,@t1) AS 't2 - t1';
+-----------+----------+
| t1 - t2   | t2 - t1  |
+-----------+----------+
| -04:30:00 | 04:30:00 |
+-----------+----------+
```

TIMEDIFF() also works for date-and-time values. That is, it accepts either time or date-and-time values, but the types of the arguments must match.

A time interval expressed as a TIME value can be broken down into components using the techniques shown in Recipe 6.8. For example, to express a time interval in terms of its constituent hours, minutes, and seconds values, calculate time interval subparts using the HOUR(), MINUTE(), and SECOND() functions. (Don't forget that if your intervals may be negative, you must take that into account.) The following SQL statement shows how to determine the components of the interval between the t1 and t2 columns of the time_val table:

```
mysql> SELECT t1, t2,
    -> TIMEDIFF(t2,t1) AS 't2 - t1 as TIME',
    -> IF(TIMEDIFF(t2,t1) >= 0,'+','-') AS sign,
    -> HOUR(TIMEDIFF(t2,t1)) AS hour,
    -> MINUTE(TIMEDIFF(t2,t1)) AS minute,
    -> SECOND(TIMEDIFF(t2,t1)) AS second
    -> FROM time_val;
+----------+----------+-----------------+------+------+--------+--------+
| t1       | t2       | t2 - t1 as TIME | sign | hour | minute | second |
+----------+----------+-----------------+------+------+--------+--------+
| 15:00:00 | 15:00:00 | 00:00:00        | +    |    0 |      0 |      0 |
| 05:01:30 | 02:30:20 | -02:31:10       | -    |    2 |     31 |     10 |
| 12:30:20 | 17:30:45 | 05:00:25        | +    |    5 |      0 |     25 |
+----------+----------+-----------------+------+------+--------+--------+
```

If you work with date or date-and-time values, the TIMESTAMPDIFF() function provides another way to calculate intervals. It enables you to specify the units in which intervals should be expressed:

```
TIMESTAMPDIFF(unit,val1,val2)
```

unit is the interval unit and *val1* and *val2* are the values between which to calculate the interval. With TIMESTAMPDIFF(), you can express an interval in many different ways:

```
mysql> SET @dt1 = '1900-01-01 00:00:00', @dt2 = '1910-01-01 00:00:00';
mysql> SELECT
    -> TIMESTAMPDIFF(MINUTE,@dt1,@dt2) AS minutes,
    -> TIMESTAMPDIFF(HOUR,@dt1,@dt2) AS hours,
    -> TIMESTAMPDIFF(DAY,@dt1,@dt2) AS days,
    -> TIMESTAMPDIFF(WEEK,@dt1,@dt2) AS weeks,
```

```
    -> TIMESTAMPDIFF(YEAR,@dt1,@dt2) AS years;
+---------+-------+------+-------+-------+
| minutes | hours | days | weeks | years |
+---------+-------+------+-------+-------+
| 5258880 | 87648 | 3652 |   521 |    10 |
+---------+-------+------+-------+-------+
```

Permitted *unit* specifiers are `MICROSECOND`, `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `QUARTER`, or `YEAR`. Note that each is singular, not plural.

Be aware of these properties of `TIMESTAMPDIFF()`:

- Its value is negative if the first temporal value is greater than the second, which is opposite the order of the arguments for `DATEDIFF()` and `TIMEDIFF()`.
- Despite the `TIMESTAMP` in its name, `TIMESTAMPDIFF()` arguments are not limited to the range of the `TIMESTAMP` data type.

Calculating intervals using basic units

Another strategy for calculating intervals is to work with basic units such as seconds or days:

1. Convert your temporal values to basic units.
2. Take the difference between the values to calculate the interval, also in basic units.
3. If you want the result as a temporal value, convert it from basic units to the appropriate type.

The conversion functions involved in implementing this strategy depend on the types of the values between which you calculate the interval:

- To convert between time values and seconds, use `TIME_TO_SEC()` and `SEC_TO_TIME()`.
- To convert between date values and days, use `TO_DAYS()` and `FROM_DAYS()`.
- To convert between date-and-time values and seconds, use `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()`.

Recipe 6.10 discusses those conversion functions (and limitations on their applicability). The following material assumes familiarity with that discussion.

Time interval calculation using basic units

To calculate intervals in seconds between pairs of time values, convert them to seconds with `TIME_TO_SEC()` and take the difference. To express the resulting interval as a `TIME` value, pass it to `SEC_TO_TIME()`. The following statement calculates the intervals be-

tween the t1 and t2 columns of the time_val table, expressing each interval both in seconds and as a TIME value:

```
mysql> SELECT t1, t2,
    -> TIME_TO_SEC(t2) - TIME_TO_SEC(t1) AS 't2 - t1 (in seconds)',
    -> SEC_TO_TIME(TIME_TO_SEC(t2) - TIME_TO_SEC(t1)) AS 't2 - t1 (as TIME)'
    -> FROM time_val;
+----------+----------+---------------------+------------------+
| t1       | t2       | t2 - t1 (in seconds) | t2 - t1 (as TIME) |
+----------+----------+---------------------+------------------+
| 15:00:00 | 15:00:00 |                   0 | 00:00:00          |
| 05:01:30 | 02:30:20 |               -9070 | -02:31:10         |
| 12:30:20 | 17:30:45 |               18025 | 05:00:25          |
+----------+----------+---------------------+------------------+
```

Date or date-and-time interval calculation using basic units

When you calculate an interval between dates by converting both dates to a common unit in relation to a given reference point and take the difference, the range of your values determines which conversions are available:

- DATE, DATETIME, or TIMESTAMP values dating back to 1970-01-01 00:00:00 UTC—the Unix epoch—can be converted to seconds elapsed since the epoch. With dates in that range, you can calculate intervals to an accuracy of one second.
- Older dates from the beginning of the Gregorian calendar (1582) on can be converted to day values and used to compute intervals in days.
- Dates that begin earlier than either of these reference points present more of a problem. In such cases, you may find that your programming language offers computations that are not available or are difficult to perform in SQL. If so, consider processing date values directly from within your API language. For example, the Date::Calc and Date::Manip modules are available from CPAN for use in Perl scripts.

To calculate an interval in days between date or date-and-time values, convert them to days with TO_DAYS() and take the difference. For an interval in weeks, do the same thing and divide the result by seven:

```
mysql> SET @days = TO_DAYS('1884-01-01') - TO_DAYS('1883-06-05');
mysql> SELECT @days AS days, @days/7 AS weeks;
+------+---------+
| days | weeks   |
+------+---------+
|  210 | 30.0000 |
+------+---------+
```

You cannot convert days to months or years by simple division because those units vary in length. To yield date intervals expressed in those units, use TIMESTAMPDIFF(), discussed earlier in this recipe.

For date-and-time values occurring within the TIMESTAMP range from 1970 partially through 2038, you can determine intervals to a resolution in seconds using the UNIX_TIMESTAMP() function. For intervals in other units, seconds are easily converted to minutes, hours, days, or weeks, as this expression shows for dates that lie two weeks apart:

```
mysql> SET @dt1 = '1984-01-01 09:00:00';
mysql> SET @dt2 = @dt1 + INTERVAL 14 DAY;
mysql> SET @interval = UNIX_TIMESTAMP(@dt2) - UNIX_TIMESTAMP(@dt1);
mysql> SELECT @interval AS seconds,
    -> @interval / 60 AS minutes,
    -> @interval / (60 * 60) AS hours,
    -> @interval / (24 * 60 * 60) AS days,
    -> @interval / (7 * 24 * 60 * 60) AS weeks;
+---------+------------+----------+---------+--------+
| seconds | minutes    | hours    | days    | weeks  |
+---------+------------+----------+---------+--------+
| 1209600 | 20160.0000 | 336.0000 | 14.0000 | 2.0000 |
+---------+------------+----------+---------+--------+
```

Use FLOOR() on the division results if you prefer integer values that have no fractional part.

For values that occur outside the TIMESTAMP range, this interval calculation method is more general (but messier):

1. Take the difference in days between the date parts of the values and multiply by 24 × 60 × 60 to convert to seconds.
2. Adjust the result by the difference in seconds between the time parts of the values.

Here's an example, using two date-and-time values that lie slightly less than three days apart:

```
mysql> SET @dt1 = '1800-02-14 07:30:00';
mysql> SET @dt2 = '1800-02-17 06:30:00';
mysql> SET @interval =
    ->   ((TO_DAYS(@dt2) - TO_DAYS(@dt1)) * 24*60*60)
    ->   + TIME_TO_SEC(@dt2) - TIME_TO_SEC(@dt1);
mysql> SELECT @interval AS seconds, SEC_TO_TIME(@interval) AS TIME;
+---------+----------+
| seconds | TIME     |
+---------+----------+
|  255600 | 71:00:00 |
+---------+----------+
```

Do You Want an Interval or a Span?

When you take a difference between dates (or times), consider whether you want an interval or a span. Taking a difference between dates gives you the interval from one date to the next. To determine the range spanned by the two dates, you must add a unit. For example, it's a three-day interval from `2002-01-01` to `2002-01-04`, but together they span a range of four days. If you don't get the results you expect from a difference-of-values calculation, consider whether an "off-by-one" correction is needed.

6.12. Adding Date or Time Values

Problem

You want to add temporal values. For example, you want to add a given number of seconds to a time or determine what the date will be three weeks from today.

Solution

To add date or time values, you have several options:

- Use one of the temporal-addition functions.
- Use the `+ INTERVAL` or `- INTERVAL` operator.
- Convert the values to basic units, and take the sum.

The applicable functions or operators depend on the types of the values.

Discussion

The following discussion shows several ways to add temporal values.

Adding temporal values using temporal-addition functions or operators

To add a time to a time or date-and-time value, use the `ADDTIME()` function:

```
mysql> SET @t1 = '12:00:00', @t2 = '15:30:00';
mysql> SELECT ADDTIME(@t1,@t2);
+------------------+
| ADDTIME(@t1,@t2) |
+------------------+
| 27:30:00         |
+------------------+
mysql> SET @dt = '1984-03-01 12:00:00', @t = '12:00:00';
mysql> SELECT ADDTIME(@dt,@t);
+---------------------+
```

```
| ADDTIME(@dt,@t)     |
+---------------------+
| 1984-03-02 00:00:00 |
+---------------------+
```

To add a time to a date or date-and-time value, use the `TIMESTAMP()` function:

```
mysql> SET @d = '1984-03-01', @t = '15:30:00';
mysql> SELECT TIMESTAMP(@d,@t);
+---------------------+
| TIMESTAMP(@d,@t)    |
+---------------------+
| 1984-03-01 15:30:00 |
+---------------------+
mysql> SET @dt = '1984-03-01 12:00:00', @t = '12:00:00';
mysql> SELECT TIMESTAMP(@dt,@t);
+---------------------+
| TIMESTAMP(@dt,@t)   |
+---------------------+
| 1984-03-02 00:00:00 |
+---------------------+
```

MySQL also provides `DATE_ADD()` and `DATE_SUB()` functions for adding intervals to dates and subtracting intervals from dates. Each function takes a date (or date-and-time) value `d` and an interval, expressed using the following syntax:

```
DATE_ADD(d,INTERVAL val unit)
DATE_SUB(d,INTERVAL val unit)
```

The `+ INTERVAL` and `- INTERVAL` operators are similar:

```
d + INTERVAL val unit
d - INTERVAL val unit
```

`unit` is the interval unit and *`val`* is an expression indicating the number of units. Some common unit specifiers are `SECOND`, `MINUTE`, `HOUR`, `DAY`, `MONTH`, and `YEAR`. Note that each is singular, not plural. (Check the *MySQL Reference Manual* for the full list.)

Use `DATE_ADD()` or `DATE_SUB()` to perform date arithmetic operations such as these:

- Determine the date three days from today:

```
mysql> SELECT CURDATE(), DATE_ADD(CURDATE(),INTERVAL 3 DAY);
+------------+-----------------------------------+
| CURDATE()  | DATE_ADD(CURDATE(),INTERVAL 3 DAY) |
+------------+-----------------------------------+
| 2014-02-20 | 2014-02-23                        |
+------------+-----------------------------------+
```

- Find the date a week ago:

```
mysql> SELECT CURDATE(), DATE_SUB(CURDATE(),INTERVAL 1 WEEK);
+------------+-----------------------------------+
| CURDATE()  | DATE_SUB(CURDATE(),INTERVAL 1 WEEK) |
+------------+-----------------------------------+
```

```
| 2014-02-20 | 2014-02-13                          |
+------------+-------------------------------------+
```

- For questions where you need to know both the date and the time, begin with a `DATETIME` or `TIMESTAMP` value. To answer the question, "What time will it be in 60 hours?" do this:

```
mysql> SELECT NOW(), DATE_ADD(NOW(),INTERVAL 60 HOUR);
+---------------------+---------------------------------+
| NOW()               | DATE_ADD(NOW(),INTERVAL 60 HOUR) |
+---------------------+---------------------------------+
| 2014-02-20 18:07:06 | 2014-02-23 06:07:06             |
+---------------------+---------------------------------+
```

- Some interval specifiers have both date and time parts. The following adds 14.5 hours to the current date and time:

```
mysql> SELECT NOW(), DATE_ADD(NOW(),INTERVAL '14:30' HOUR_MINUTE);
+---------------------+-------------------------------------------+
| NOW()               | DATE_ADD(NOW(),INTERVAL '14:30' HOUR_MINUTE) |
+---------------------+-------------------------------------------+
| 2014-02-20 18:07:06 | 2014-02-21 08:37:06                       |
+---------------------+-------------------------------------------+
```

  Similarly, adding 3 days and 4 hours produces this result:

```
mysql> SELECT NOW(), DATE_ADD(NOW(),INTERVAL '3 4' DAY_HOUR);
+---------------------+--------------------------------------+
| NOW()               | DATE_ADD(NOW(),INTERVAL '3 4' DAY_HOUR) |
+---------------------+--------------------------------------+
| 2014-02-20 18:07:06 | 2014-02-23 22:07:06                  |
+---------------------+--------------------------------------+
```

`DATE_ADD()` and `DATE_SUB()` are interchangeable because one is the same as the other with the sign of the interval value flipped. These two expressions are equivalent for any date value `d`:

```
DATE_ADD(d,INTERVAL -3 MONTH)
DATE_SUB(d,INTERVAL 3 MONTH)
```

You can also use the `+ INTERVAL` or `- INTERVAL` operator to perform date interval addition or subtraction:

```
mysql> SELECT CURDATE(), CURDATE() + INTERVAL 1 YEAR;
+------------+---------------------------+
| CURDATE()  | CURDATE() + INTERVAL 1 YEAR |
+------------+---------------------------+
| 2014-02-20 | 2015-02-20                |
+------------+---------------------------+
mysql> SELECT NOW(), NOW() - INTERVAL '1 12' DAY_HOUR;
+---------------------+---------------------------------+
| NOW()               | NOW() - INTERVAL '1 12' DAY_HOUR |
+---------------------+---------------------------------+
```

```
| 2014-02-20 18:07:06 | 2014-02-19 06:07:06             |
+---------------------+---------------------------------+
```

`TIMESTAMPADD()` is an alternative function for adding intervals to date or date-and-time values. Its arguments are similar to those for `DATE_ADD()`, and the following equivalence holds:

```
TIMESTAMPADD(unit,interval,d) = DATE_ADD(d,INTERVAL interval unit)
```

Adding temporal values using basic units

Another way to add intervals to date or date-and-time values is to perform temporal "shifting" via functions that convert to and from basic units. For background information about the applicable functions, see Recipe 6.10.

Adding time values using basic units

Adding times with basic units is similar to calculating intervals between times, except that you compute a sum rather than a difference. To add an interval value in seconds to a `TIME` value, convert the `TIME` to seconds so that both values are represented in the same units, add the values and convert the result back to a `TIME`. For example, two hours is 7,200 seconds ($2 \times 60 \times 60$), so the following statement adds two hours to each `t1` value in the `time_val` table:

```
mysql> SELECT t1,
    -> SEC_TO_TIME(TIME_TO_SEC(t1) + 7200) AS 't1 plus 2 hours'
    -> FROM time_val;
+----------+-----------------+
| t1       | t1 plus 2 hours |
+----------+-----------------+
| 15:00:00 | 17:00:00        |
| 05:01:30 | 07:01:30        |
| 12:30:20 | 14:30:20        |
+----------+-----------------+
```

If the interval itself is expressed as a `TIME`, it too should be converted to seconds before adding the values together. The following example calculates the sum of the two `TIME` values in each row of the `time_val` table:

```
mysql> SELECT t1, t2,
    -> TIME_TO_SEC(t1) + TIME_TO_SEC(t2)
    ->   AS 't1 + t2 (in seconds)',
    -> SEC_TO_TIME(TIME_TO_SEC(t1) + TIME_TO_SEC(t2))
    ->   AS 't1 + t2 (as TIME)'
    -> FROM time_val;
+----------+----------+----------------------+------------------+
| t1       | t2       | t1 + t2 (in seconds) | t1 + t2 (as TIME) |
+----------+----------+----------------------+------------------+
| 15:00:00 | 15:00:00 |               108000 | 30:00:00          |
| 05:01:30 | 02:30:20 |                27110 | 07:31:50          |
```

```
| 12:30:20 | 17:30:45 |               108065 | 30:01:05          |
+----------+----------+----------------------+-------------------+
```

It's important to recognize that MySQL `TIME` values represent elapsed time, not time of day, so they don't reset to 0 after reaching 24 hours. You can see this in the first and third output rows from the previous statement. To produce time-of-day values, enforce a 24-hour wraparound using a modulo operation before converting the seconds value back to a `TIME` value. The number of seconds in a day is 24 × 60 × 60, or 86,400. To convert any seconds value `s` to lie within a 24-hour range, use the `MOD()` function or the `%` modulo operator like this:

```
MOD(s,86400)
s % 86400
s MOD 86400
```

The three expressions are equivalent. Applying the first of them to the time calculations from the preceding example produces the following result:

```
mysql> SELECT t1, t2,
    -> MOD(TIME_TO_SEC(t1) + TIME_TO_SEC(t2), 86400)
    ->   AS 't1 + t2 (in seconds)',
    -> SEC_TO_TIME(MOD(TIME_TO_SEC(t1) + TIME_TO_SEC(t2), 86400))
    ->   AS 't1 + t2 (as TIME)'
    -> FROM time_val;
+----------+----------+----------------------+-------------------+
| t1       | t2       | t1 + t2 (in seconds) | t1 + t2 (as TIME) |
+----------+----------+----------------------+-------------------+
| 15:00:00 | 15:00:00 |                21600 | 06:00:00          |
| 05:01:30 | 02:30:20 |                27110 | 07:31:50          |
| 12:30:20 | 17:30:45 |                21665 | 06:01:05          |
+----------+----------+----------------------+-------------------+
```

The permitted range of a `TIME` column is `-838:59:59` to `838:59:59` (that is, `-3020399` to `3020399` seconds). However, the range of `TIME` *expressions* can be greater, so when you add time values, you can easily produce a result that lies outside this range and cannot be stored as is into a `TIME` column.

Adding to date or date-and-time values using basic units

Date or date-and-time values converted to basic units can be shifted to produce other dates. For example, to shift a date forward or backward a week (seven days), use `TO_DAYS()` and `FROM_DAYS()`:

```
mysql> SET @d = '1980-01-01';
mysql> SELECT @d AS date,
    -> FROM_DAYS(TO_DAYS(@d) + 7) AS 'date + 1 week',
    -> FROM_DAYS(TO_DAYS(@d) - 7) AS 'date - 1 week';
+------------+--------------+--------------+
| date       | date + 1 week | date - 1 week |
```

```
+------------+--------------+--------------+
| 1980-01-01 | 1980-01-08   | 1979-12-25   |
+------------+--------------+--------------+
```

`TO_DAYS()` also can convert date-and-time values to days, if you don't mind having it chop off the time part.

To preserve the time, you can use `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` instead, if the initial and resulting values both lie in the permitted range for `TIMESTAMP` values (from 1970 partially through 2038). The following statement shifts a `DATETIME` value forward and backward by an hour (3,600 seconds):

```
mysql> SET @dt = '1980-01-01 09:00:00';
mysql> SELECT @dt AS datetime,
    -> FROM_UNIXTIME(UNIX_TIMESTAMP(@dt) + 3600) AS 'datetime + 1 hour',
    -> FROM_UNIXTIME(UNIX_TIMESTAMP(@dt) - 3600) AS 'datetime - 1 hour';
+---------------------+---------------------+---------------------+
| datetime            | datetime + 1 hour   | datetime - 1 hour   |
+---------------------+---------------------+---------------------+
| 1980-01-01 09:00:00 | 1980-01-01 10:00:00 | 1980-01-01 08:00:00 |
+---------------------+---------------------+---------------------+
```

6.13. Calculating Ages

Problem

You want to know how old someone is.

Solution

This is a date-arithmetic problem. It amounts to computing the interval between dates, but with a twist. For an age in years, it's necessary to account for the relative placement of the start and end dates within the calendar year. For an age in months, it's also necessary to account for the placement of the months and the days within the month.

Discussion

Age determination is a type of date-interval calculation. However, you cannot simply compute a difference in days and divide by 365 because leap years throw off the calculation. (It is 365 days from 1995-03-01 to 1996-02-29, but that is not a year in age terms.) Dividing by 365.25 is slightly more accurate, but still not correct for all dates.

To calculate ages, use the `TIMESTAMPDIFF()` function. Pass it a birth date, a current date, and the unit in which you want the age expressed:

```
TIMESTAMPDIFF(unit,birth,current)
```

`TIMESTAMPDIFF()` handles the calculations necessary to adjust for differing month and year lengths and relative positions of the dates within the calendar year. Suppose that a `sibling` table lists the birth dates of Gretchen Smith and her brothers Wilbur and Franz:

```
mysql> SELECT * FROM sibling;
+----------+------------+
| name     | birth      |
+----------+------------+
| Gretchen | 1942-04-14 |
| Wilbur   | 1946-11-28 |
| Franz    | 1953-03-05 |
+----------+------------+
```

Using `TIMESTAMPDIFF()`, you can answer questions such as these:

- How old are the Smith children today, in years and months?

```
mysql> SELECT name, birth, CURDATE() AS today,
    -> TIMESTAMPDIFF(YEAR,birth,CURDATE()) AS 'age in years',
    -> TIMESTAMPDIFF(MONTH,birth,CURDATE()) AS 'age in months'
    -> FROM sibling;
+----------+------------+------------+--------------+---------------+
| name     | birth      | today      | age in years | age in months |
+----------+------------+------------+--------------+---------------+
| Gretchen | 1942-04-14 | 2014-02-20 |           71 |           862 |
| Wilbur   | 1946-11-28 | 2014-02-20 |           67 |           806 |
| Franz    | 1953-03-05 | 2014-02-20 |           60 |           731 |
+----------+------------+------------+--------------+---------------+
```

- How old were Gretchen and Wilbur when Franz was born, in years and months?

```
mysql> SELECT name, birth, '1953-03-05' AS 'Franz'' birth',
    -> TIMESTAMPDIFF(YEAR,birth,'1953-03-05') AS 'age in years',
    -> TIMESTAMPDIFF(MONTH,birth,'1953-03-05') AS 'age in months'
    -> FROM sibling WHERE name <> 'Franz';
+----------+------------+--------------+--------------+---------------+
| name     | birth      | Franz' birth | age in years | age in months |
+----------+------------+--------------+--------------+---------------+
| Gretchen | 1942-04-14 | 1953-03-05   |           10 |           130 |
| Wilbur   | 1946-11-28 | 1953-03-05   |            6 |            75 |
+----------+------------+--------------+--------------+---------------+
```

6.14. Finding the First Day, Last Day, or Length of a Month

Problem

Given a date, you want to determine the date for the first or last day of the month in which the date occurs, or the first or last day for the month *n* months away. A related problem is to determine the number of days in a month.

Solution

To determine the date for the first day in a month, use date shifting (an application of date arithmetic). To determine the date for the last day, use the `LAST_DAY()` function. To determine the number of days in a month, find the date for its last day and use it as the argument to `DAYOFMONTH()`.

Discussion

Sometimes you have a reference date and want to reach a target date that doesn't have a fixed relationship to the reference date. For example, the first or last days of the current month aren't a fixed number of days from the current date.

To find the first day of the month for a given date, shift the date back by one fewer days than its `DAYOFMONTH()` value:

```
mysql> SELECT d, DATE_SUB(d,INTERVAL DAYOFMONTH(d)-1 DAY) AS '1st of month'
    -> FROM date_val;
+------------+--------------+
| d          | 1st of month |
+------------+--------------+
| 1864-02-28 | 1864-02-01   |
| 1900-01-15 | 1900-01-01   |
| 1999-12-31 | 1999-12-01   |
| 2000-06-04 | 2000-06-01   |
| 2017-03-16 | 2017-03-01   |
+------------+--------------+
```

In the general case, to find the first of the month for any month *n* months away from a given date, calculate the first of the month for the date and shift the result by *n* months:

```
DATE_ADD(DATE_SUB(d,INTERVAL DAYOFMONTH(d)-1 DAY),INTERVAL n MONTH)
```

For example, to find the first day of the previous and following months relative to a given date, *n* is `-1` and `1`:

```
mysql> SELECT d,
    -> DATE_ADD(DATE_SUB(d,INTERVAL DAYOFMONTH(d)-1 DAY),INTERVAL -1 MONTH)
    ->   AS '1st of previous month',
    -> DATE_ADD(DATE_SUB(d,INTERVAL DAYOFMONTH(d)-1 DAY),INTERVAL 1 MONTH)
    ->   AS '1st of following month'
    -> FROM date_val;
+------------+-----------------------+-----------------------+
| d          | 1st of previous month | 1st of following month |
+------------+-----------------------+-----------------------+
| 1864-02-28 | 1864-01-01            | 1864-03-01            |
| 1900-01-15 | 1899-12-01            | 1900-02-01            |
| 1999-12-31 | 1999-11-01            | 2000-01-01            |
| 2000-06-04 | 2000-05-01            | 2000-07-01            |
| 2017-03-16 | 2017-02-01            | 2017-04-01            |
+------------+-----------------------+-----------------------+
```

It's easier to find the last day of the month for a given date because there is a function for it:

```
mysql> SELECT d, LAST_DAY(d) AS 'last of month'
    -> FROM date_val;
+------------+---------------+
| d          | last of month |
+------------+---------------+
| 1864-02-28 | 1864-02-29    |
| 1900-01-15 | 1900-01-31    |
| 1999-12-31 | 1999-12-31    |
| 2000-06-04 | 2000-06-30    |
| 2017-03-16 | 2017-03-31    |
+------------+---------------+
```

For the general case, to find the last of the month for any month *n* months away from a given date, shift the date by that many months first, then pass it to `LAST_DAY()`:

```
LAST_DAY(DATE_ADD(d,INTERVAL n MONTH))
```

For example, to find the last day of the previous and following months relative to a given date, *n* is `-1` and `1`:

```
mysql> SELECT d,
    -> LAST_DAY(DATE_ADD(d,INTERVAL -1 MONTH))
    ->   AS 'last of previous month',
    -> LAST_DAY(DATE_ADD(d,INTERVAL 1 MONTH))
    ->  AS 'last of following month'
    -> FROM date_val;
+------------+-----------------------+------------------------+
| d          | last of previous month | last of following month |
+------------+-----------------------+------------------------+
| 1864-02-28 | 1864-01-31            | 1864-03-31             |
| 1900-01-15 | 1899-12-31            | 1900-02-28             |
| 1999-12-31 | 1999-11-30            | 2000-01-31             |
| 2000-06-04 | 2000-05-31            | 2000-07-31             |
| 2017-03-16 | 2017-02-28            | 2017-04-30             |
+------------+-----------------------+------------------------+
```

To find the length of a month in days, determine the date of its last day with `LAST_DAY()`, then use `DAYOFMONTH()` to extract the day-of-month component from the result:

```
mysql> SELECT d, DAYOFMONTH(LAST_DAY(d)) AS 'days in month' FROM date_val;
+------------+--------------+
| d          | days in month |
+------------+--------------+
| 1864-02-28 |           29 |
| 1900-01-15 |           31 |
| 1999-12-31 |           31 |
| 2000-06-04 |           30 |
| 2017-03-16 |           31 |
+------------+--------------+
```

See Also

Recipe 6.18 discusses how to calculate month lengths from within a program without using SQL. (The trick is that you must account for leap years.)

6.15. Calculating Dates by Substring Replacement

Problem

Given a date, you want to produce another date from it when you know that the two dates share some components in common.

Solution

Treat a date or time value as a string, and perform direct replacement on parts of the string.

Discussion

In some cases, you can use substring replacement to calculate dates without performing any date arithmetic. For example, a string operation produces the first-of-month value for a given date by replacing the day component with `01`. You can do this either with `DATE_FORMAT()` or with `CONCAT()`:

```
mysql> SELECT d,
    -> DATE_FORMAT(d,'%Y-%m-01') AS '1st of month A',
    -> CONCAT(YEAR(d),'-',LPAD(MONTH(d),2,'0'),'-01') AS '1st of month B'
    -> FROM date_val;
+------------+---------------+---------------+
| d          | 1st of month A | 1st of month B |
+------------+---------------+---------------+
| 1864-02-28 | 1864-02-01     | 1864-02-01     |
| 1900-01-15 | 1900-01-01     | 1900-01-01     |
| 1999-12-31 | 1999-12-01     | 1999-12-01     |
| 2000-06-04 | 2000-06-01     | 2000-06-01     |
| 2017-03-16 | 2017-03-01     | 2017-03-01     |
+------------+---------------+---------------+
```

The string replacement technique can also produce dates with a specific position within the calendar year. For New Year's Day (January 1), replace the month and day with `01`; for Christmas, replace the month and day with `12` and `25`:

```
mysql> SELECT d,
    -> DATE_FORMAT(d,'%Y-01-01') AS 'New Year''s A',
    -> CONCAT(YEAR(d),'-01-01') AS 'New Year''s B',
    -> DATE_FORMAT(d,'%Y-12-25') AS 'Christmas A',
    -> CONCAT(YEAR(d),'-12-25') AS 'Christmas B'
    -> FROM date_val;
```

```
+------------+-------------+-------------+-------------+-------------+
| d          | New Year's A | New Year's B | Christmas A | Christmas B |
+------------+-------------+-------------+-------------+-------------+
| 1864-02-28 | 1864-01-01   | 1864-01-01   | 1864-12-25  | 1864-12-25  |
| 1900-01-15 | 1900-01-01   | 1900-01-01   | 1900-12-25  | 1900-12-25  |
| 1999-12-31 | 1999-01-01   | 1999-01-01   | 1999-12-25  | 1999-12-25  |
| 2000-06-04 | 2000-01-01   | 2000-01-01   | 2000-12-25  | 2000-12-25  |
| 2017-03-16 | 2017-01-01   | 2017-01-01   | 2017-12-25  | 2017-12-25  |
+------------+-------------+-------------+-------------+-------------+
```

To perform the same operation for the target calendar day in other years, combine string replacement with date shifting. The following statement shows two ways to determine the date for Christmas two years hence. The first method finds Christmas for this year, and then shifts it two years forward. The second shifts the current date forward two years, and then finds Christmas in the resulting year:

```
mysql> SELECT CURDATE(),
    -> DATE_ADD(DATE_FORMAT(CURDATE(),'%Y-12-25'),INTERVAL 2 YEAR)
    ->   AS 'Christmas A',
    -> DATE_FORMAT(DATE_ADD(CURDATE(),INTERVAL 2 YEAR),'%Y-12-25')
    ->   AS 'Christmas B';
+------------+-------------+-------------+
| CURDATE()  | Christmas A | Christmas B |
+------------+-------------+-------------+
| 2014-02-20 | 2016-12-25  | 2016-12-25  |
+------------+-------------+-------------+
```

6.16. Finding the Day of the Week for a Date

Problem

You want to know the day of the week on which a date falls.

Solution

Use the `DAYNAME()` function.

Discussion

To determine the name of the day of the week for a given date, use `DAYNAME()`:

```
mysql> SELECT CURDATE(), DAYNAME(CURDATE());
+------------+--------------------+
| CURDATE()  | DAYNAME(CURDATE()) |
+------------+--------------------+
| 2014-02-20 | Thursday           |
+------------+--------------------+
```

DAYNAME() is often useful in conjunction with other date-related techniques. For example, to determine the day of the week for the first of the month, use the first-of-month expression from Recipe 6.14 as the argument to DAYNAME():

```
mysql> SET @d = CURDATE();
mysql> SET @first = DATE_SUB(@d,INTERVAL DAYOFMONTH(@d)-1 DAY);
mysql> SELECT @d AS 'starting date',
    -> @first AS '1st of month date',
    -> DAYNAME(@first) AS '1st of month day';
+---------------+-------------------+------------------+
| starting date | 1st of month date | 1st of month day |
+---------------+-------------------+------------------+
| 2014-02-20    | 2014-02-01        | Saturday         |
+---------------+-------------------+------------------+
```

6.17. Finding Dates for Any Weekday of a Given Week

Problem

You want to compute the date of some weekday for the week in which a given date lies. Suppose that you want to know the date of the Tuesday that falls in the same week as 2014-07-09.

Solution

This is an application of date shifting. Figure out the number of days between the starting weekday of the given date and the desired day, and shift the date by that many days.

Discussion

This section and the next describe how to convert one date to another when the target date is specified in terms of days of the week. To solve such problems, you need to know day-of-week values. Suppose you begin with a target date of 2014-07-09. To determine the date for Tuesday of the week in which that date lies, the calculation depends on what weekday it is. If it's a Monday, you add a day to produce 2014-07-10, but if it's a Wednesday, you subtract a day to produce 2014-07-08.

MySQL provides two functions that are useful here. DAYOFWEEK() treats Sunday as the first day of the week and returns 1 through 7 for Sunday through Saturday. WEEKDAY() treats Monday as the first day of the week and returns 0 through 6 for Monday through Sunday. (The examples shown here use DAYOFWEEK().) Another kind of day-of-week operation involves determining the name of the day. DAYNAME() can be used for that.

Calculations that determine one day of the week from another depend on the day you start from as well as the day you want to reach. I find it easiest to shift the reference date first to a known point relative to the beginning of the week, and then shift forward:

1. Shift the reference date back by its DAYOFWEEK() value, which always produces the date for the Saturday preceding the week.
2. Shift the Saturday date by one day to reach the Sunday date, by two days to reach the Monday date, and so forth.

In SQL, those operations can be expressed as follows for a date d, where *n* is 1 through 7 to produce the dates for Sunday through Saturday:

```
DATE_ADD(DATE_SUB(d,INTERVAL DAYOFWEEK(d) DAY),INTERVAL n DAY)
```

That expression splits the "shift back to Saturday" and "shift forward" phases into separate operations, but because the intervals for both DATE_SUB() and DATE_ADD() are in days, the expression can be simplified into a single DATE_ADD() call:

```
DATE_ADD(d,INTERVAL n-DAYOFWEEK(d) DAY)
```

Applying this formula to the dates in our date_val table, using an *n* of 1 for Sunday and 7 for Saturday to find the first and last days of the week, yields this result:

```
mysql> SELECT d, DAYNAME(d) AS day,
    -> DATE_ADD(d,INTERVAL 1-DAYOFWEEK(d) DAY) AS Sunday,
    -> DATE_ADD(d,INTERVAL 7-DAYOFWEEK(d) DAY) AS Saturday
    -> FROM date_val;
+------------+----------+------------+------------+
| d          | day      | Sunday     | Saturday   |
+------------+----------+------------+------------+
| 1864-02-28 | Sunday   | 1864-02-28 | 1864-03-05 |
| 1900-01-15 | Monday   | 1900-01-14 | 1900-01-20 |
| 1999-12-31 | Friday   | 1999-12-26 | 2000-01-01 |
| 2000-06-04 | Sunday   | 2000-06-04 | 2000-06-10 |
| 2017-03-16 | Thursday | 2017-03-12 | 2017-03-18 |
+------------+----------+------------+------------+
```

To determine the date of some weekday in a week relative to that of the target date, modify the preceding procedure a bit. First, determine the date of the desired weekday in the target date. Then shift the result into the desired week.

Calculating the date for a day of the week in some other week is a problem that breaks down into a day-within-week shift (using the formula just given) plus a week shift. These operations can be done in either order because the amount of shift within the week is the same whether or not you shift the reference date into a different week first. For example, to calculate Wednesday of a week by the preceding formula, *n* is 4. To compute the date for Wednesday two weeks ago, you can perform the day-within-week shift first, like this:

```
mysql> SET @target =
    -> DATE_SUB(DATE_ADD(CURDATE(),INTERVAL 4-DAYOFWEEK(CURDATE()) DAY),
    -> INTERVAL 14 DAY);
mysql> SELECT CURDATE(), @target, DAYNAME(@target);
```

```
+------------+------------+------------------+
| CURDATE()  | @target    | DAYNAME(@target) |
+------------+------------+------------------+
| 2014-02-20 | 2014-02-05 | Wednesday        |
+------------+------------+------------------+
```

Or you can perform the week shift first:

```
mysql> SET @target =
    -> DATE_ADD(DATE_SUB(CURDATE(),INTERVAL 14 DAY),
    -> INTERVAL 4-DAYOFWEEK(CURDATE()) DAY);
mysql> SELECT CURDATE(), @target, DAYNAME(@target);
+------------+------------+------------------+
| CURDATE()  | @target    | DAYNAME(@target) |
+------------+------------+------------------+
| 2014-02-20 | 2014-02-05 | Wednesday        |
+------------+------------+------------------+
```

Some applications need to determine dates such as the *n*-th instance of particular weekdays. For example, to administer a payroll for which paydays are the second and fourth Thursdays of each month, you must know what those dates are. One way to do this for any given month is to begin with the first-of-month date and shift it forward. It's easy enough to shift the date to the Thursday in that week; the trick is to figure out how many weeks forward to shift the result to reach the second and fourth Thursdays. If the first of the month occurs on any day from Sunday through Thursday, you shift forward one week to reach the second Thursday. If the first of the month occurs on Friday or later, you shift forward by two weeks. The fourth Thursday is, of course, two weeks after that.

The following Perl code implements this logic to find all paydays in the year 2014. It runs a loop that constructs the first-of-month date for the months of the year. For each month, it issues a statement that determines the dates of the second and fourth Thursdays:

```
my $year = 2014;
print "MM/CCYY   2nd Thursday   4th Thursday\n";
foreach my $month (1..12)
{
  my $first = sprintf ("%04d-%02d-01", $year, $month);
  my ($thu2, $thu4) = $dbh->selectrow_array (qq{
                  SELECT
                    DATE_ADD(
                      DATE_ADD(?,INTERVAL 5-DAYOFWEEK(?) DAY),
                      INTERVAL IF(DAYOFWEEK(?) <= 5, 7, 14) DAY),
                    DATE_ADD(
                      DATE_ADD(?,INTERVAL 5-DAYOFWEEK(?) DAY),
                      INTERVAL IF(DAYOFWEEK(?) <= 5, 21, 28) DAY)
                }, undef, $first, $first, $first, $first, $first, $first);
  printf "%02d/%04d   %s     %s\n", $month, $year, $thu2, $thu4;
}
```

The program produces this output:

```
MM/CCYY   2nd Thursday   4th Thursday
01/2014   2014-01-09     2014-01-23
02/2014   2014-02-13     2014-02-27
03/2014   2014-03-13     2014-03-27
04/2014   2014-04-10     2014-04-24
05/2014   2014-05-08     2014-05-22
06/2014   2014-06-12     2014-06-26
07/2014   2014-07-10     2014-07-24
08/2014   2014-08-14     2014-08-28
09/2014   2014-09-11     2014-09-25
10/2014   2014-10-09     2014-10-23
11/2014   2014-11-13     2014-11-27
12/2014   2014-12-11     2014-12-25
```

6.18. Performing Leap-Year Calculations

Problem

A date calculation must account for leap years. For example, the length of a month or a year depends on whether the date falls in a leap year.

Solution

Know how to test whether a year is a leap year, and factor the result into your calculation.

Discussion

Date calculations are complicated by the fact that months differ in length. An additional twist is that February has an extra day during leap years. This recipe shows how to determine whether any given date falls within a leap year and how to take leap years into account when determining the length of a year or month.

Determining whether a date occurs in a leap year

To determine whether a date `d` falls within a leap year, obtain the year component using `YEAR()` and test the result. The common rule-of-thumb test for leap years is "divisible by four," which you can test using a modulo operation:

```
YEAR(d) % 4 = 0
```

However, that test is not technically correct. (For example, the year 1900 is divisible by four, but is *not* a leap year.) For a year to qualify as a leap year, it must satisfy both of these constraints:

- The year must be divisible by four.
- The year cannot be divisible by 100, unless it is also divisible by 400.

The meaning of the second constraint is that turn-of-century years are not leap years, except every fourth century. In SQL, express these conditions as follows:

```
(YEAR(d) % 4 = 0) AND ((YEAR(d) % 100 <> 0) OR (YEAR(d) % 400 = 0))
```

Running our `date_val` table through both the rule-of-thumb leap-year test and the complete test produces the following results:

```
mysql> SELECT
    -> d,
    -> YEAR(d) % 4 = 0
    ->   AS 'rule-of-thumb test',
    -> (YEAR(d) % 4 = 0) AND ((YEAR(d) % 100 <> 0) OR (YEAR(d) % 400 = 0))
    ->   AS 'complete test'
    -> FROM date_val;
+------------+--------------------+---------------+
| d          | rule-of-thumb test | complete test |
+------------+--------------------+---------------+
| 1864-02-28 |                  1 |             1 |
| 1900-01-15 |                  1 |             0 |
| 1999-12-31 |                  0 |             0 |
| 2000-06-04 |                  1 |             1 |
| 2017-03-16 |                  0 |             0 |
+------------+--------------------+---------------+
```

As you can see, results from the two tests sometimes differ. In particular, the rule-of-thumb test fails for the year 1900; the complete test result is correct because it accounts for the turn-of-century constraint.

Because the complete leap-year test must check the century, it requires four-digit year values. Two-digit years are ambiguous with respect to the century, making it impossible to assess the turn-of-century constraint.

To make the leap-year test easier to perform in SQL statements, use a stored function that encapsulates the expression just shown. The *routines* directory of the `recipes` distribution contains a script that creates an `is_leap_year()` function.

If you work with date values within a program, you can perform leap-year tests with your API language rather than at the SQL level. Extract the first four digits of the date string to get the year, then test it. If the language performs automatic string-to-number conversion of the year value, this is easy. Otherwise, you must explicitly convert the year value to numeric form before testing it.

Perl, PHP:

```
$year = substr ($date, 0, 4);
$is_leap = ($year % 4 == 0) && ($year % 100 != 0 || $year % 400 == 0);
```

Ruby:

```
year = date[0..3].to_i
is_leap = (year.modulo(4) == 0) &&
            (year.modulo(100) != 0 || year.modulo(400) == 0)
```

Python:

```
year = int(date[0:4])
is_leap = (year % 4 == 0) and (year % 100 != 0 or year % 400 == 0)
```

Java:

```
int year = Integer.valueOf (date.substring (0, 4)).intValue ();
boolean is_leap = (year % 4 == 0) && (year % 100 != 0 || year % 400 == 0);
```

Your API language might provide its own means of determining leap years. For example, the PHP `date()` function has an `L` option to return whether a date falls in a leap year:

```
# prevent date () from complaining about not knowing time zone
date_default_timezone_set ("UTC");
$is_leap = date ("L", strtotime ($date));
```

Using leap-year tests for year-length calculations

Years usually have 365 days, but leap years have 366. To determine the length of a year in which a date falls, use one of the leap-year tests just shown to figure out whether to add a day. This example uses Perl:

```
$year = substr ($date, 0, 4);
$is_leap = ($year % 4 == 0) && ($year % 100 != 0 || $year % 400 == 0);
$days_in_year = ($is_leap ? 366 : 365);
```

To compute a year's length in SQL, compute the date of the last day of the year and pass it to `DAYOFYEAR()`:

```
mysql> SET @d1 = '2014-04-13', @d2 = '2016-04-13';
mysql> SELECT
    -> DAYOFYEAR(DATE_FORMAT(@d1,'%Y-12-31')) AS 'days in 2014',
    -> DAYOFYEAR(DATE_FORMAT(@d2,'%Y-12-31')) AS 'days in 2016';
+--------------+--------------+
| days in 2014 | days in 2016 |
+--------------+--------------+
|          365 |          366 |
+--------------+--------------+
```

Using leap-year tests for month-length calculations

Recipe 6.14 discusses how to determine the number of days in a month in SQL statements using the `LAST_DAY()` function. Within an API language, you can write a non–SQL-based function that, given an ISO-format date argument, returns the number of days in the month during which the date occurs. This is straightforward except for

February, where the function must return 29 or 28 depending on whether the year is a leap year. Here's a Ruby version:

```
def days_in_month(date)
  year = date[0..3].to_i
  month = date[5..6].to_i   # month, 1-based
  days_in_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  days = days_in_month[month-1]
  is_leap = (year.modulo(4) == 0) &&
              (year.modulo(100) != 0 || year.modulo(400) == 0)

  # add a day for Feb of leap years
  days += 1 if month == 2 && is_leap
  return days
end
```

See Also

Recipe 12.11 discusses leap-year calculations in the context of date validation.

6.19. Canonizing Not-Quite-ISO Date Strings

Problem

A date is in a format that's close to but not exactly ISO format.

Solution

Canonize the date by passing it to a function that always returns an ISO-format date result.

Discussion

In Recipe 6.9, we ran into the problem that synthesizing dates with `CONCAT()` may produce values that are not quite in ISO format. For example, the following statement produces first-of-month values in which the month part may have only a single digit:

```
mysql> SELECT d, CONCAT(YEAR(d),'-',MONTH(d),'-01') FROM date_val;
+------------+-----------------------------------+
| d          | CONCAT(YEAR(d),'-',MONTH(d),'-01') |
+------------+-----------------------------------+
| 1864-02-28 | 1864-2-01                         |
| 1900-01-15 | 1900-1-01                         |
| 1999-12-31 | 1999-12-01                        |
| 2000-06-04 | 2000-6-01                         |
| 2017-03-16 | 2017-3-01                         |
+------------+-----------------------------------+
```

Recipe 6.9 shows a technique using `LPAD()` for making sure the month values have two digits. Another way to standardize a close-to-ISO date is to use it in an expression that produces an ISO date result. For a date `d`, any of the following expressions will do:

```
DATE_ADD(d,INTERVAL 0 DAY)
d + INTERVAL 0 DAY
FROM_DAYS(TO_DAYS(d))
STR_TO_DATE(d,'%Y-%m-%d')
```

Using those expressions with the non-ISO results from the `CONCAT()` operation yields ISO format in several ways:

```
mysql> SELECT
    -> CONCAT(YEAR(d),'-',MONTH(d),'-01') AS 'non-ISO',
    -> DATE_ADD(CONCAT(YEAR(d),'-',MONTH(d),'-01'),INTERVAL 0 DAY) AS 'ISO 1',
    -> CONCAT(YEAR(d),'-',MONTH(d),'-01') + INTERVAL 0 DAY AS 'ISO 2',
    -> FROM_DAYS(TO_DAYS(CONCAT(YEAR(d),'-',MONTH(d),'-01'))) AS 'ISO 3',
    -> STR_TO_DATE(CONCAT(YEAR(d),'-',MONTH(d),'-01'),'%Y-%m-%d') AS 'ISO 4'
    -> FROM date_val;
+------------+------------+------------+------------+------------+
| non-ISO    | ISO 1      | ISO 2      | ISO 3      | ISO 4      |
+------------+------------+------------+------------+------------+
| 1864-2-01  | 1864-02-01 | 1864-02-01 | 1864-02-01 | 1864-02-01 |
| 1900-1-01  | 1900-01-01 | 1900-01-01 | 1900-01-01 | 1900-01-01 |
| 1999-12-01 | 1999-12-01 | 1999-12-01 | 1999-12-01 | 1999-12-01 |
| 2000-6-01  | 2000-06-01 | 2000-06-01 | 2000-06-01 | 2000-06-01 |
| 2017-3-01  | 2017-03-01 | 2017-03-01 | 2017-03-01 | 2017-03-01 |
+------------+------------+------------+------------+------------+
```

6.20. Selecting Rows Based on Temporal Characteristics

Problem

You want to select rows based on temporal conditions.

Solution

Use a date or time condition in the `WHERE` clause. This may be based on direct comparison of column values with known values. Or it may be necessary to apply a function to column values to convert them to a more appropriate form for testing, such as using `MONTH()` to test the month part of a date.

Discussion

Most of the preceding date-based techniques were illustrated by example statements that produce date or time values as output. To place date-based restrictions on the rows selected by a statement, use the same techniques in a `WHERE` clause. For example, you

can select rows by looking for values that occur before or after a given date, within a date range, or that match particular month or day values.

Comparing dates to one another

The following statements find rows from the `date_val` table that occur either before 1900 or during the 1900s:

```
mysql> SELECT d FROM date_val where d < '1900-01-01';
+------------+
| d          |
+------------+
| 1864-02-28 |
+------------+
mysql> SELECT d FROM date_val where d BETWEEN '1900-01-01' AND '1999-12-31';
+------------+
| d          |
+------------+
| 1900-01-15 |
| 1999-12-31 |
+------------+
```

When you don't know the exact date needed for a comparison in a `WHERE` clause, you can often calculate it using an expression. For example, to perform an "on this day in history" statement to search for rows in a table named `history` to find events occurring exactly 50 years ago, do this:

```
SELECT * FROM history WHERE d = DATE_SUB(CURDATE(),INTERVAL 50 YEAR);
```

You see this kind of thing in newspapers that run columns showing what the news events were in times past. (In essence, the statement identifies those events that have reached their *n*-th anniversary.) To retrieve events that occurred "on this day" for any year rather than "on this date" for a specific year, the statement is a bit different. In that case, you need to find rows that match the current calendar day, ignoring the year. That topic is discussed in "Comparing dates to calendar days" on page 231.

Calculated dates are useful for range testing as well. For example, to find dates that occur later than 20 years ago, use `DATE_SUB()` to calculate the cutoff date:

```
mysql> SELECT d FROM date_val WHERE d >= DATE_SUB(CURDATE(),INTERVAL 20 YEAR);
+------------+
| d          |
+------------+
| 1999-12-31 |
| 2000-06-04 |
| 2017-03-16 |
+------------+
```

Note that the expression in the `WHERE` clause isolates the date column `d` on one side of the comparison operator. This is usually a good idea; if the column is indexed, placing it alone on one side of a comparison enables MySQL to process the statement more

efficiently. To illustrate, the preceding `WHERE` clause can be written in a way that's logically equivalent but much less efficient for MySQL to execute:

```
WHERE DATE_ADD(d,INTERVAL 20 YEAR) >= CURDATE();
```

Here, the `d` column is used within an expression. That means *every* row must be retrieved so that the expression can be evaluated and tested, which makes any index on the column useless.

Sometimes it's not so obvious how to rewrite a comparison to isolate a date column on one side. For example, the following `WHERE` clause uses only part of the date column in the comparisons:

```
WHERE YEAR(d) >= 1987 AND YEAR(d) <= 1991;
```

To rewrite the first comparison, eliminate the `YEAR()` call, and replace its right side with a complete date:

```
WHERE d >= '1987-01-01' AND YEAR(d) <= 1991;
```

Rewriting the second comparison is a little trickier. You can eliminate the `YEAR()` call on the left side, just as with the first expression, but you can't just add `-01-01` to the year on the right side. That produces the following result, which is incorrect:

```
WHERE d >= '1987-01-01' AND d <= '1991-01-01';
```

That fails because dates from `1991-01-02` to `1991-12-31` fail the test, but should pass. To rewrite the second comparison correctly, do this:

```
WHERE d >= '1987-01-01' AND d < '1992-01-01';
```

Another use for calculated dates occurs frequently in applications that create rows that have a limited lifetime. Such applications must be able to determine which rows to delete when performing an expiration operation. You can approach this problem a couple ways:

- Store a date in each row indicating when it was created. (Do this by making the column a `TIMESTAMP` or by setting it to `NOW()`; see Recipe 6.7 for details.) To perform an expiration operation later, determine which rows have a creation date that is too old by comparing that date to the current date. For example, the statement to expire rows that were created more than *n* days ago might look like this:

  ```
  DELETE FROM mytbl WHERE create_date < DATE_SUB(NOW(),INTERVAL n DAY);
  ```

- Store an explicit expiration date in each row by calculating the expiration date with `DATE_ADD()` when the row is created. For a row that should expire in *n* days, do this:

  ```
  INSERT INTO mytbl (expire_date,...)
  VALUES(DATE_ADD(NOW(),INTERVAL n DAY),...);
  ```

  To perform the expiration operation in this case, compare the expiration dates to the current date to see which have been reached:

```
DELETE FROM mytbl WHERE expire_date < NOW();
```

Comparing times to one another

Comparisons involving times are similar to those involving dates. For example, to find times in the `t1` column that occurred from 9 AM to 2 PM, use an expression like one of these:

```
WHERE t1 BETWEEN '09:00:00' AND '14:00:00';
WHERE HOUR(t1) BETWEEN 9 AND 14;
```

For an indexed `TIME` column, the first method is more efficient. The second method has the property that it works not only for `TIME` columns, but for `DATETIME` and `TIMESTAMP` columns as well.

Comparing dates to calendar days

To answer questions about particular days of the year, use calendar-day testing. The following examples illustrate how to do this in the context of looking for birthdays:

- Who has a birthday today? This requires matching a particular calendar day, so you extract the month and day but ignore the year when performing comparisons:

  ```
  WHERE MONTH(d) = MONTH(CURDATE()) AND DAYOFMONTH(d) = DAYOFMONTH(CURDATE());
  ```

  This kind of statement commonly is applied to biographical data to find lists of actors, politicians, musicians, and so forth, who were born on a particular day of the year.

  It's tempting to use `DAYOFYEAR()` to solve "on this day" problems because it results in simpler statements. But `DAYOFYEAR()` doesn't work properly for leap years. The presence of February 29 throws off the values for days from March through December.

- Who has a birthday this month? In this case, it's necessary to check only the month:

  ```
  WHERE MONTH(d) = MONTH(CURDATE());
  ```

- Who has a birthday next month? The trick here is that you can't just add one to the current month to get the month number that qualifying dates must match. That gives you 13 for dates in December. To make sure that you get 1 (January), use either of the following techniques:

  ```
  WHERE MONTH(d) = MONTH(DATE_ADD(CURDATE(),INTERVAL 1 MONTH));
  WHERE MONTH(d) = MOD(MONTH(CURDATE()),12)+1;
  ```

CHAPTER 7

Sorting Query Results

7.0. Introduction

This chapter covers sorting, an extremely important operation for controlling how MySQL displays results from `SELECT` statements. To sort a query result, add an `ORDER BY` clause to the query. Without such a clause, MySQL is free to return rows in any order, so sorting helps bring order to disorder and makes query results easier to examine and understand.

You can sort rows of a query result several ways:

- Using a single column, a combination of columns, or even parts of columns or expression results
- Using ascending or descending order
- Using case-sensitive or case-insensitive string comparisons
- Using temporal ordering

Several examples in this chapter use the `driver_log` table, which contains columns for recording daily mileage logs for a set of truck drivers:

```
mysql> SELECT * FROM driver_log;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      1 | Ben   | 2014-07-30 |   152 |
|      2 | Suzi  | 2014-07-29 |   391 |
|      3 | Henry | 2014-07-29 |   300 |
|      4 | Henry | 2014-07-27 |    96 |
|      5 | Ben   | 2014-07-29 |   131 |
|      6 | Henry | 2014-07-26 |   115 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      8 | Henry | 2014-08-01 |   197 |
```

```
|      9 | Ben   | 2014-08-02 |    79 |
|     10 | Henry | 2014-07-30 |   203 |
+--------+-------+------------+-------+
```

Many other examples use the `mail` table (used in earlier chapters):

```
mysql> SELECT * FROM mail;
+---------------------+---------+---------+---------+---------+---------+
| t                   | srcuser | srchost | dstuser | dsthost | size    |
+---------------------+---------+---------+---------+---------+---------+
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |   58274 |
| 2014-05-12 12:48:13 | tricia  | mars    | gene    | venus   |  194925 |
| 2014-05-12 15:02:49 | phil    | mars    | phil    | saturn  |    1048 |
| 2014-05-12 18:59:18 | barb    | saturn  | tricia  | venus   |     271 |
| 2014-05-14 09:31:37 | gene    | venus   | barb    | mars    |    2291 |
| 2014-05-14 11:52:17 | phil    | mars    | tricia  | saturn  |    5781 |
| 2014-05-14 14:42:21 | barb    | venus   | barb    | venus   |   98151 |
| 2014-05-14 17:03:01 | tricia  | saturn  | phil    | venus   | 2394482 |
| 2014-05-15 07:17:48 | gene    | mars    | gene    | saturn  |    3824 |
| 2014-05-15 08:50:57 | phil    | venus   | phil    | venus   |     978 |
| 2014-05-15 10:25:52 | gene    | mars    | tricia  | saturn  |  998532 |
| 2014-05-15 17:35:31 | gene    | saturn  | gene    | mars    |    3856 |
| 2014-05-16 09:00:28 | gene    | venus   | barb    | mars    |     613 |
| 2014-05-16 23:04:19 | phil    | venus   | barb    | venus   |   10294 |
| 2014-05-19 12:49:23 | phil    | mars    | tricia  | saturn  |     873 |
| 2014-05-19 22:21:51 | gene    | saturn  | gene    | venus   |   23992 |
+---------------------+---------+---------+---------+---------+---------+
```

Other tables are used occasionally as well. To create them, use scripts found in the *tables* directory of the `recipes` distribution.

7.1. Using ORDER BY to Sort Query Results

Problem

Rows in a query result don't appear in the order you want.

Solution

Add an `ORDER BY` clause to the query to sort its result.

Discussion

The contents of the `driver_log` and `mail` tables shown in the chapter introduction are disorganized and difficult to make sense of. The exception is that the values in the `id` and `t` columns are in order, but that's just coincidental. Rows do tend to be returned from a table in the order in which they were originally inserted, but only until the table is subjected to delete and update operations. Rows inserted after that are likely to be returned in the middle of the result set somewhere. Many MySQL users notice this

disturbance in row-retrieval order, which leads them to ask, "How can I store rows in my table so they come out in a particular order when I retrieve them?" The answer to this question is, "That's the wrong question." Storing rows is the server's job, and you should let the server do it. Even if you could specify storage order, it wouldn't help you if you want results in different orders at different times.

When you select rows, they're returned from the database in whatever order the server happens to use. A relational database makes no guarantee about the order in which it returns rows—unless you tell it how, by adding an `ORDER BY` clause to your `SELECT` statement. Without `ORDER BY`, you may find that the retrieval order changes over time as you modify the table contents. With an `ORDER BY` clause, MySQL always sorts rows as you indicate.

`ORDER BY` has the following general characteristics:

- You can sort using one or more column or expression values.
- You can sort columns independently in ascending order (the default) or descending order.
- You can refer to sort columns by name or by using an alias.

This section shows some basic sorting techniques, such as how to name the sort columns and specify the sort direction. The following sections illustrate how to perform more complex sorts. Paradoxically, you can even use `ORDER BY` to *disorder* a result set, which is useful for randomizing the rows or (in conjunction with `LIMIT`) for picking a row at random from a result set (see Recipes 15.7 and 15.8).

The following examples demonstrate how to sort on a single column or multiple columns and how to sort in ascending or descending order. The examples select the rows in the `driver_log` table but sort them in different orders to demonstrate the effect of the different `ORDER BY` clauses.

This query produces a single-column sort using the driver name:

```
mysql> SELECT * FROM driver_log ORDER BY name;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      1 | Ben   | 2014-07-30 |   152 |
|      9 | Ben   | 2014-08-02 |    79 |
|      5 | Ben   | 2014-07-29 |   131 |
|      8 | Henry | 2014-08-01 |   197 |
|      6 | Henry | 2014-07-26 |   115 |
|      4 | Henry | 2014-07-27 |    96 |
|      3 | Henry | 2014-07-29 |   300 |
|     10 | Henry | 2014-07-30 |   203 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      2 | Suzi  | 2014-07-29 |   391 |
+--------+-------+------------+-------+
```

The default sort direction is ascending. To make the direction for an ascending sort explicit, add ASC after the sorted column's name:

```
SELECT * FROM driver_log ORDER BY name ASC;
```

The opposite (or reverse) of ascending order is descending order, specified by adding DESC after the sorted column's name:

```
mysql> SELECT * FROM driver_log ORDER BY name DESC;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      2 | Suzi  | 2014-07-29 |   391 |
|      7 | Suzi  | 2014-08-02 |   502 |
|     10 | Henry | 2014-07-30 |   203 |
|      8 | Henry | 2014-08-01 |   197 |
|      6 | Henry | 2014-07-26 |   115 |
|      4 | Henry | 2014-07-27 |    96 |
|      3 | Henry | 2014-07-29 |   300 |
|      5 | Ben   | 2014-07-29 |   131 |
|      9 | Ben   | 2014-08-02 |    79 |
|      1 | Ben   | 2014-07-30 |   152 |
+--------+-------+------------+-------+
```

Closely examine the output from the queries just shown and you'll notice that although rows are sorted by name, rows for any given name are in no special order. (The trav_date values aren't in date order for Henry or Ben, for example.) That's because MySQL doesn't sort something unless you tell it to:

- The overall order of rows returned by a query is indeterminate unless you specify an ORDER BY clause.
- Within a group of rows that sort together based on the values in a given column, the order of values in other columns also is indeterminate unless you name them in the ORDER BY clause.

To more fully control output order, specify a multiple-column sort by listing each column to use for sorting, separated by commas. The following query sorts in ascending order by name and by trav_date within the rows for each name:

```
mysql> SELECT * FROM driver_log ORDER BY name, trav_date;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      5 | Ben   | 2014-07-29 |   131 |
|      1 | Ben   | 2014-07-30 |   152 |
|      9 | Ben   | 2014-08-02 |    79 |
|      6 | Henry | 2014-07-26 |   115 |
|      4 | Henry | 2014-07-27 |    96 |
|      3 | Henry | 2014-07-29 |   300 |
|     10 | Henry | 2014-07-30 |   203 |
```

```
|      8 | Henry | 2014-08-01 |   197 |
|      2 | Suzi  | 2014-07-29 |   391 |
|      7 | Suzi  | 2014-08-02 |   502 |
+--------+-------+------------+-------+
```

Multiple-column sorts can be descending as well, but `DESC` must be specified after *each* column name to perform a fully descending sort.

Multiple-column `ORDER BY` clauses can perform mixed-order sorting where some columns are sorted in ascending order and others in descending order. The following query sorts by `name` in descending order, then by `trav_date` in ascending order for each name:

```
mysql> SELECT * FROM driver_log ORDER BY name DESC, trav_date;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      2 | Suzi  | 2014-07-29 |   391 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      6 | Henry | 2014-07-26 |   115 |
|      4 | Henry | 2014-07-27 |    96 |
|      3 | Henry | 2014-07-29 |   300 |
|     10 | Henry | 2014-07-30 |   203 |
|      8 | Henry | 2014-08-01 |   197 |
|      5 | Ben   | 2014-07-29 |   131 |
|      1 | Ben   | 2014-07-30 |   152 |
|      9 | Ben   | 2014-08-02 |    79 |
+--------+-------+------------+-------+
```

The `ORDER BY` clauses in the queries shown thus far refer to the sorted columns by name. You can also name the columns by using aliases. That is, if an output column has an alias, you can refer to the alias in the `ORDER BY` clause:

```
mysql> SELECT name, trav_date, miles AS distance FROM driver_log
    -> ORDER BY distance;
+-------+------------+----------+
| name  | trav_date  | distance |
+-------+------------+----------+
| Ben   | 2014-08-02 |       79 |
| Henry | 2014-07-27 |       96 |
| Henry | 2014-07-26 |      115 |
| Ben   | 2014-07-29 |      131 |
| Ben   | 2014-07-30 |      152 |
| Henry | 2014-08-01 |      197 |
| Henry | 2014-07-30 |      203 |
| Henry | 2014-07-29 |      300 |
| Suzi  | 2014-07-29 |      391 |
| Suzi  | 2014-08-02 |      502 |
+-------+------------+----------+
```

7.2. Using Expressions for Sorting

Problem

You want to sort a query result based on values calculated from a column rather than the values actually stored in the column.

Solution

Put the expression that calculates the values in the `ORDER BY` clause.

Discussion

One of the `mail` table columns shows how large each mail message is, in bytes:

```
mysql> SELECT * FROM mail;
+---------------------+---------+---------+---------+---------+---------+
| t                   | srcuser | srchost | dstuser | dsthost | size    |
+---------------------+---------+---------+---------+---------+---------+
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |   58274 |
| 2014-05-12 12:48:13 | tricia  | mars    | gene    | venus   |  194925 |
| 2014-05-12 15:02:49 | phil    | mars    | phil    | saturn  |    1048 |
| 2014-05-12 18:59:18 | barb    | saturn  | tricia  | venus   |     271 |
…
```

Suppose that you want to retrieve rows for "big" mail messages (defined as those larger than 50,000 bytes), but you want them to be displayed and sorted by sizes in terms of kilobytes, not bytes. In this case, the values to sort are calculated by an expression:

```
FLOOR((size+1023)/1024)
```

The `+1023` in the `FLOOR()` expression groups `size` values to the nearest upper boundary of the 1,024-byte categories. Without it, the values group by lower boundaries (for example, a 2,047-byte message is reported as having a size of 1 kilobyte rather than 2). Recipe 8.10 disscusses this technique in more detail.

To sort by that expression, put it directly in the `ORDER BY` clause:

```
mysql> SELECT t, srcuser, FLOOR((size+1023)/1024)
    -> FROM mail WHERE size > 50000
    -> ORDER BY FLOOR((size+1023)/1024);
+---------------------+---------+------------------------+
| t                   | srcuser | FLOOR((size+1023)/1024) |
+---------------------+---------+------------------------+
| 2014-05-11 10:15:08 | barb    |                     57 |
| 2014-05-14 14:42:21 | barb    |                     96 |
| 2014-05-12 12:48:13 | tricia  |                    191 |
| 2014-05-15 10:25:52 | gene    |                    976 |
| 2014-05-14 17:03:01 | tricia  |                   2339 |
+---------------------+---------+------------------------+
```

Alternatively, if the sorting expression appears in the output column list, you can alias it there and refer to the alias in the ORDER BY clause:

```
mysql> SELECT t, srcuser, FLOOR((size+1023)/1024) AS kilobytes
    -> FROM mail WHERE size > 50000
    -> ORDER BY kilobytes;
+---------------------+---------+-----------+
| t                   | srcuser | kilobytes |
+---------------------+---------+-----------+
| 2014-05-11 10:15:08 | barb    |        57 |
| 2014-05-14 14:42:21 | barb    |        96 |
| 2014-05-12 12:48:13 | tricia  |       191 |
| 2014-05-15 10:25:52 | gene    |       976 |
| 2014-05-14 17:03:01 | tricia  |      2339 |
+---------------------+---------+-----------+
```

You might prefer the alias method for several reasons:

- It's easier to write the alias in the ORDER BY clause than to repeat the (cumbersome) expression.
- Without the alias, if you change the expression one place, you must change it in the other.
- The alias may be useful for display purposes, to provide a better column label. Note how the third column heading is much more meaningful in the second of the two preceding queries.

7.3. Displaying One Set of Values While Sorting by Another

Problem

You want to sort a result set using values that don't appear in the output column list.

Solution

That's not a problem. The ORDER BY clause can refer to columns you don't display.

Discussion

ORDER BY is not limited to sorting only those columns named in the output column list. It can sort using values that are "hidden" (that is, not displayed in the query output). This technique is commonly used when you have values that can be represented different ways and you want to display one type of value but sort by another. For example, you may want to display mail message sizes not in terms of bytes, but as strings such as

103K for 103 kilobytes. You can convert a byte count to that kind of value using this expression:

```
CONCAT(FLOOR((size+1023)/1024),'K')
```

However, such values are strings, so they sort lexically, not numerically. If you use them for sorting, a value such as 96K sorts after 2339K, even though it represents a smaller number:

```
mysql> SELECT t, srcuser,
    -> CONCAT(FLOOR((size+1023)/1024),'K') AS size_in_K
    -> FROM mail WHERE size > 50000
    -> ORDER BY size_in_K;
+---------------------+---------+-----------+
| t                   | srcuser | size_in_K |
+---------------------+---------+-----------+
| 2014-05-12 12:48:13 | tricia  | 191K      |
| 2014-05-14 17:03:01 | tricia  | 2339K     |
| 2014-05-11 10:15:08 | barb    | 57K       |
| 2014-05-14 14:42:21 | barb    | 96K       |
| 2014-05-15 10:25:52 | gene    | 976K      |
+---------------------+---------+-----------+
```

To achieve the desired output order, display the string, but use actual numeric size for sorting:

```
mysql> SELECT t, srcuser,
    -> CONCAT(FLOOR((size+1023)/1024),'K') AS size_in_K
    -> FROM mail WHERE size > 50000
    -> ORDER BY size;
+---------------------+---------+-----------+
| t                   | srcuser | size_in_K |
+---------------------+---------+-----------+
| 2014-05-11 10:15:08 | barb    | 57K       |
| 2014-05-14 14:42:21 | barb    | 96K       |
| 2014-05-12 12:48:13 | tricia  | 191K      |
| 2014-05-15 10:25:52 | gene    | 976K      |
| 2014-05-14 17:03:01 | tricia  | 2339K     |
+---------------------+---------+-----------+
```

Displaying values as strings but sorting them as numbers helps solve some otherwise difficult problems. Members of sports teams typically are assigned a jersey number, which normally you might think should be stored using a numeric column. Not so fast! Some players like to have a jersey number of zero (0), and some like double-zero (00). If a team happens to have players with both numbers, you cannot represent them using a numeric column because both values will be treated as the same number. To solve this problem, store jersey numbers as strings:

```
CREATE TABLE roster
(
  name        CHAR(30),   # player name
```

```
  jersey_num  CHAR(3)     # jersey number
);
```

Then the jersey numbers will display the same way you enter them, and `0` and `00` will be treated as distinct values. Unfortunately, although representing numbers as strings solves the problem of distinguishing `0` and `00`, it introduces a different problem. Suppose that a team has the following players:

```
mysql> SELECT name, jersey_num FROM roster;
+-----------+------------+
| name      | jersey_num |
+-----------+------------+
| Lynne     | 29         |
| Ella      | 0          |
| Elizabeth | 100        |
| Nancy     | 00         |
| Jean      | 8          |
| Sherry    | 47         |
+-----------+------------+
```

Now try to sort the team members by jersey number. If those numbers are stored as strings, they sort lexically, and lexical order often differs from numeric order. That's certainly true for the team in question:

```
mysql> SELECT name, jersey_num FROM roster ORDER BY jersey_num;
+-----------+------------+
| name      | jersey_num |
+-----------+------------+
| Ella      | 0          |
| Nancy     | 00         |
| Elizabeth | 100        |
| Lynne     | 29         |
| Sherry    | 47         |
| Jean      | 8          |
+-----------+------------+
```

The values `100` and `8` are out of place, but that's easily solved: display the string values and use the numeric values for sorting. To accomplish this, add zero to the `jersey_num` values to force a string-to-number conversion:

```
mysql> SELECT name, jersey_num FROM roster ORDER BY jersey_num+0;
+-----------+------------+
| name      | jersey_num |
+-----------+------------+
| Ella      | 0          |
| Nancy     | 00         |
| Jean      | 8          |
| Lynne     | 29         |
| Sherry    | 47         |
| Elizabeth | 100        |
+-----------+------------+
```

The technique of displaying one value but sorting by another is also useful when you display values composed from multiple columns that don't sort the way you want. For example, the `mail` table lists message senders using separate `srcuser` and `srchost` values. To display message senders from the `mail` table as email addresses in `srcuser@srchost` format with the username first, construct those values using the following expression:

```
CONCAT(srcuser,'@',srchost)
```

However, those values are no good for sorting if you want to treat the hostname as more significant than the username. Instead, sort the results using the underlying column values rather than the displayed composite values:

```
mysql> SELECT t, CONCAT(srcuser,'@',srchost) AS sender, size
    -> FROM mail WHERE size > 50000
    -> ORDER BY srchost, srcuser;
+---------------------+---------------+---------+
| t                   | sender        | size    |
+---------------------+---------------+---------+
| 2014-05-15 10:25:52 | gene@mars     |  998532 |
| 2014-05-12 12:48:13 | tricia@mars   |  194925 |
| 2014-05-11 10:15:08 | barb@saturn   |   58274 |
| 2014-05-14 17:03:01 | tricia@saturn | 2394482 |
| 2014-05-14 14:42:21 | barb@venus    |   98151 |
+---------------------+---------------+---------+
```

The same idea commonly applies to sorting people's names. Suppose that a `names` table contains last and first names. To display rows sorted by last name first, the query is straightforward when the columns are displayed separately:

```
mysql> SELECT last_name, first_name FROM name
    -> ORDER BY last_name, first_name;
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Blue      | Vida       |
| Brown     | Kevin      |
| Gray      | Pete       |
| White     | Devon      |
| White     | Rondell    |
+-----------+------------+
```

If instead you want to display each name as a single string composed of the first name, a space, and the last name, begin the query like this:

```
SELECT CONCAT(first_name,' ',last_name) AS full_name FROM name ...
```

But then how do you sort the names so they come out in last-name order? Display composite names, but refer to the constituent values in the `ORDER BY` clause:

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS full_name
    -> FROM name
```

```
    -> ORDER BY last_name, first_name;
+---------------+
| full_name     |
+---------------+
| Vida Blue     |
| Kevin Brown   |
| Pete Gray     |
| Devon White   |
| Rondell White |
+---------------+
```

7.4. Controlling Case Sensitivity of String Sorts

Problem

String-sorting operations are case sensitive when you don't want them to be, or vice versa.

Solution

Alter the comparison characteristics of the sorted values.

Discussion

Recipe 5.1 discusses how string-comparison properties depend on whether the strings are binary or nonbinary:

- Binary strings are sequences of bytes. They are compared byte by byte using numeric byte values. Character set and lettercase have no meaning for comparisons.
- Nonbinary strings are sequences of characters. They have a character set and collation and are compared character by character using the order defined by the collation.

These properties also apply to string sorting because sorting is based on comparison. To alter the sorting properties of a string column, alter its comparison properties. (For a summary of which string data types are binary and nonbinary, see Recipe 5.2.)

The examples in this section use a table that has case-insensitive and case-sensitive nonbinary columns, and a binary column:

```
CREATE TABLE str_val
(
  ci_str   CHAR(3) CHARACTER SET latin1 COLLATE latin1_swedish_ci,
  cs_str   CHAR(3) CHARACTER SET latin1 COLLATE latin1_general_cs,
  bin_str  BINARY(3)
);
```

Suppose that the table has these contents:

```
+--------+--------+---------+
| ci_str | cs_str | bin_str |
+--------+--------+---------+
| AAA    | AAA    | AAA     |
| aaa    | aaa    | aaa     |
| bbb    | bbb    | bbb     |
| BBB    | BBB    | BBB     |
+--------+--------+---------+
```

Each column contains the same values, but the natural sort orders for the column data types produce three different results:

- The case-insensitive collation sorts `a` and `A` together, placing them before `b` and `B`. However, for a given letter, it does not necessarily order one lettercase before another, as shown by the following result:

```
mysql> SELECT ci_str FROM str_val ORDER BY ci_str;
+--------+
| ci_str |
+--------+
| AAA    |
| aaa    |
| bbb    |
| BBB    |
+--------+
```

- The case-sensitive collation puts `A` and `a` before `B` and `b`, and sorts uppercase before lowercase:

```
mysql> SELECT cs_str FROM str_val ORDER BY cs_str;
+--------+
| cs_str |
+--------+
| AAA    |
| aaa    |
| BBB    |
| bbb    |
+--------+
```

- The binary strings sort numerically. Assuming that uppercase letters have numeric values less than those of lowercase letters, a binary sort results in the following ordering:

```
mysql> SELECT bin_str FROM str_val ORDER BY bin_str;
+---------+
| bin_str |
+---------+
| AAA     |
| BBB     |
| aaa     |
```

```
| bbb     |
+---------+
```

You get the same result for a nonbinary string column that has a binary collation, as long as the column contains single-byte characters (for example, `CHAR(3) CHARACTER SET latin1 COLLATE latin1_bin`). For multibyte characters, a binary collation still produces a numeric sort, but the character values use multibyte numbers.

To alter the sorting properties of each column, use the techniques described in Recipe 5.7 for controlling string comparisons:

- To sort case-insensitive strings in case-sensitive fashion, order the sorted values using a case-sensitive collation:

```
mysql> SELECT ci_str FROM str_val
    -> ORDER BY ci_str COLLATE latin1_general_cs;
+--------+
| ci_str |
+--------+
| AAA    |
| aaa    |
| BBB    |
| bbb    |
+--------+
```

- To sort case-sensitive strings in case-insensitive fashion, order the sorted values using a case-insensitive collation:

```
mysql> SELECT cs_str FROM str_val
    -> ORDER BY cs_str COLLATE latin1_swedish_ci;
+--------+
| cs_str |
+--------+
| AAA    |
| aaa    |
| bbb    |
| BBB    |
+--------+
```

  Alternatively, sort using values that have been converted to the same lettercase, which makes lettercase irrelevant:

```
mysql> SELECT cs_str FROM str_val
    -> ORDER BY UPPER(cs_str);
+--------+
| cs_str |
+--------+
| AAA    |
| aaa    |
| bbb    |
| BBB    |
+--------+
```

- Binary strings sort using numeric byte values, so there is no concept of lettercase involved. However, because letters in different cases have different byte values, comparisons of binary strings effectively are case sensitive. (That is, `a` and `A` are unequal.) To sort binary strings using a case-insensitive ordering, convert them to nonbinary strings and apply an appropriate collation. For example, to perform a case-insensitive sort, use a statement like this:

```
mysql> SELECT bin_str FROM str_val
    -> ORDER BY CONVERT(bin_str USING latin1) COLLATE latin1_swedish_ci;
+---------+
| bin_str |
+---------+
| AAA     |
| aaa     |
| bbb     |
| BBB     |
+---------+
```

  If the character-set default collation is case insensitive (as is true for `latin1`), you can omit the `COLLATE` clause.

7.5. Date-Based Sorting

Problem

You want to sort rows in temporal order.

Solution

Sort using a date or time column. If some parts of the values are irrelevant for the sort that you want to accomplish, ignore them.

Discussion

Many database tables include date or time information and it's very often necessary to sort results in temporal order. MySQL knows how to sort temporal data types, so there's no special trick to ordering them. The next few examples use the `mail` table, which contains a `DATETIME` column, but the same sorting principles apply to `DATE`, `TIME`, and `TIMESTAMP` columns.

Here are the messages sent by `phil`:

```
mysql> SELECT * FROM mail WHERE srcuser = 'phil';
+---------------------+---------+---------+---------+---------+-------+
| t                   | srcuser | srchost | dstuser | dsthost | size  |
+---------------------+---------+---------+---------+---------+-------+
| 2014-05-12 15:02:49 | phil    | mars    | phil    | saturn  |  1048 |
```

```
| 2014-05-14 11:52:17 | phil    | mars    | tricia  | saturn  |  5781 |
| 2014-05-15 08:50:57 | phil    | venus   | phil    | venus   |   978 |
| 2014-05-16 23:04:19 | phil    | venus   | barb    | venus   | 10294 |
| 2014-05-19 12:49:23 | phil    | mars    | tricia  | saturn  |   873 |
+---------------------+---------+---------+---------+---------+-------+
```

To display the messages, most recently sent ones first, use `ORDER BY` with `DESC`:

```
mysql> SELECT * FROM mail WHERE srcuser = 'phil' ORDER BY t DESC;
+---------------------+---------+---------+---------+---------+-------+
| t                   | srcuser | srchost | dstuser | dsthost | size  |
+---------------------+---------+---------+---------+---------+-------+
| 2014-05-19 12:49:23 | phil    | mars    | tricia  | saturn  |   873 |
| 2014-05-16 23:04:19 | phil    | venus   | barb    | venus   | 10294 |
| 2014-05-15 08:50:57 | phil    | venus   | phil    | venus   |   978 |
| 2014-05-14 11:52:17 | phil    | mars    | tricia  | saturn  |  5781 |
| 2014-05-12 15:02:49 | phil    | mars    | phil    | saturn  |  1048 |
+---------------------+---------+---------+---------+---------+-------+
```

Sometimes a temporal sort uses only part of a date or time column. In that case, use an expression that extracts the part or parts you need and sort the result using the expression. Some examples of this are given in the following discussion.

Sorting by time of day

You can do time-of-day sorting different ways, depending on your column type. If the values are stored in a `TIME` column named `timecol`, just sort them directly using `ORDER BY timecol`. To put `DATETIME` or `TIMESTAMP` values in time-of-day order, extract the time parts and sort them. For example, the `mail` table contains `DATETIME` values, which can be sorted by time of day like this:

```
mysql> SELECT * FROM mail ORDER BY TIME(t);
+---------------------+---------+---------+---------+---------+---------+
| t                   | srcuser | srchost | dstuser | dsthost | size    |
+---------------------+---------+---------+---------+---------+---------+
| 2014-05-15 07:17:48 | gene    | mars    | gene    | saturn  |    3824 |
| 2014-05-15 08:50:57 | phil    | venus   | phil    | venus   |     978 |
| 2014-05-16 09:00:28 | gene    | venus   | barb    | mars    |     613 |
| 2014-05-14 09:31:37 | gene    | venus   | barb    | mars    |    2291 |
| 2014-05-11 10:15:08 | barb    | saturn  | tricia  | mars    |   58274 |
| 2014-05-15 10:25:52 | gene    | mars    | tricia  | saturn  |  998532 |
| 2014-05-14 11:52:17 | phil    | mars    | tricia  | saturn  |    5781 |
| 2014-05-12 12:48:13 | tricia  | mars    | gene    | venus   |  194925 |
…
```

Sorting by calendar day

To sort date values in calendar order, ignore the year part of the dates and use only the month and day to order values by where they fall during the calendar year. Suppose that an `occasion` table looks like this when values are ordered by date:

```
mysql> SELECT date, description FROM occasion ORDER BY date;
+------------+-----------------------------------+
| date       | description                       |
+------------+-----------------------------------+
| 1215-06-15 | Signing of the Magna Carta        |
| 1732-02-22 | George Washington's birthday      |
| 1776-07-14 | Bastille Day                      |
| 1789-07-04 | US Independence Day               |
| 1809-02-12 | Abraham Lincoln's birthday        |
| 1919-06-28 | Signing of the Treaty of Versailles |
| 1944-06-06 | D-Day at Normandy Beaches         |
| 1957-10-04 | Sputnik launch date               |
| 1989-11-09 | Opening of the Berlin Wall        |
+------------+-----------------------------------+
```

To put these items in calendar order, sort them by month and day within month:

```
mysql> SELECT date, description FROM occasion
    -> ORDER BY MONTH(date), DAYOFMONTH(date);
+------------+-----------------------------------+
| date       | description                       |
+------------+-----------------------------------+
| 1809-02-12 | Abraham Lincoln's birthday        |
| 1732-02-22 | George Washington's birthday      |
| 1944-06-06 | D-Day at Normandy Beaches         |
| 1215-06-15 | Signing of the Magna Carta        |
| 1919-06-28 | Signing of the Treaty of Versailles |
| 1789-07-04 | US Independence Day               |
| 1776-07-14 | Bastille Day                      |
| 1957-10-04 | Sputnik launch date               |
| 1989-11-09 | Opening of the Berlin Wall        |
+------------+-----------------------------------+
```

MySQL has a `DAYOFYEAR()` function that you might suspect would be useful for calendar-day sorting. However, it can generate the same value for different calendar days. For example, February 29 of leap years and March 1 of nonleap years have the same day-of-year value:

```
mysql> SELECT DAYOFYEAR('1996-02-29'), DAYOFYEAR('1997-03-01');
+-------------------------+-------------------------+
| DAYOFYEAR('1996-02-29') | DAYOFYEAR('1997-03-01') |
+-------------------------+-------------------------+
|                      60 |                      60 |
+-------------------------+-------------------------+
```

This means that `DAYOFYEAR()` can group dates that actually occur on different calendar days.

If a table represents dates using separate year, month, and day columns, calendar sorting requires no date-part extraction. Just sort the relevant columns directly. For large datasets, sorting using separate date-part columns can be much faster than sorts based on extracting pieces of `DATE` values. There's no overhead for part extraction, but more

importantly, you can index the date-part columns separately—something not possible with a DATE column. The principle here is that you should design the table to make it easy to extract or sort by the values that you expect to use a lot.

Sorting by day of week

Day-of-week sorting is similar to calendar-day sorting, except that you use different functions to obtain the relevant ordering values.

You can get the day of the week using DAYNAME(), but that produces strings that sort lexically rather than in day-of-week order (Sunday, Monday, Tuesday, and so forth). Here the technique of displaying one value but sorting by another is useful (see Recipe 7.3). Display day names using DAYNAME(), but sort in day-of-week order using DAYOFWEEK(), which returns numeric values from 1 to 7 for Sunday through Saturday:

```
mysql> SELECT DAYNAME(date) AS day, date, description
    -> FROM occasion
    -> ORDER BY DAYOFWEEK(date);
+----------+------------+-----------------------------------+
| day      | date       | description                       |
+----------+------------+-----------------------------------+
| Sunday   | 1776-07-14 | Bastille Day                      |
| Sunday   | 1809-02-12 | Abraham Lincoln's birthday        |
| Monday   | 1215-06-15 | Signing of the Magna Carta        |
| Tuesday  | 1944-06-06 | D-Day at Normandy Beaches         |
| Thursday | 1989-11-09 | Opening of the Berlin Wall        |
| Friday   | 1957-10-04 | Sputnik launch date               |
| Friday   | 1732-02-22 | George Washington's birthday      |
| Saturday | 1789-07-04 | US Independence Day               |
| Saturday | 1919-06-28 | Signing of the Treaty of Versailles |
+----------+------------+-----------------------------------+
```

To sort rows in day-of-week order but treat Monday as the first day of the week and Sunday as the last, use the MOD() function to map Monday to 0, Tuesday to 1, …, Sunday to 6:

```
mysql> SELECT DAYNAME(date), date, description
    -> FROM occasion
    -> ORDER BY MOD(DAYOFWEEK(date)+5, 7);
+---------------+------------+-----------------------------------+
| DAYNAME(date) | date       | description                       |
+---------------+------------+-----------------------------------+
| Monday        | 1215-06-15 | Signing of the Magna Carta        |
| Tuesday       | 1944-06-06 | D-Day at Normandy Beaches         |
| Thursday      | 1989-11-09 | Opening of the Berlin Wall        |
| Friday        | 1957-10-04 | Sputnik launch date               |
| Friday        | 1732-02-22 | George Washington's birthday      |
| Saturday      | 1789-07-04 | US Independence Day               |
| Saturday      | 1919-06-28 | Signing of the Treaty of Versailles |
| Sunday        | 1776-07-14 | Bastille Day                      |
```

```
| Sunday        | 1809-02-12 | Abraham Lincoln's birthday          |
+---------------+------------+-------------------------------------+
```

The following table shows the `DAYOFWEEK()` expressions for putting any day of the week first in the sort order:

Day to list first	DAYOFWEEK() expression
Sunday	`DAYOFWEEK(date)`
Monday	`MOD(DAYOFWEEK(date)+5, 7)`
Tuesday	`MOD(DAYOFWEEK(date)+4, 7)`
Wednesday	`MOD(DAYOFWEEK(date)+3, 7)`
Thursday	`MOD(DAYOFWEEK(date)+2, 7)`
Friday	`MOD(DAYOFWEEK(date)+1, 7)`
Saturday	`MOD(DAYOFWEEK(date)+0, 7)`

You can also use `WEEKDAY()` for day-of-week sorting, although it returns a different set of values (0 for Monday through 6 for Sunday).

7.6. Sorting by Substrings of Column Values

Problem

You want to sort a set of values using one or more substrings of each value.

Solution

Extract the pieces you want and sort them separately.

Discussion

This is a specific application of sorting by expression value (see Recipe 7.2). To sort rows using just a particular portion of a column's values, extract the substring you need and use it in the `ORDER BY` clause. This is easiest if the substrings are at a fixed position and length within the column. For substrings of variable position or length, you can still use them for sorting if you have a reliable way to identify them. The next several recipes show how to use substring extraction to produce specialized sort orders.

7.7. Sorting by Fixed-Length Substrings

Problem

You want to sort using parts of a column that occur at a given position within the column.

Solution

Pull out the parts you need with `LEFT()`, `MID()`, or `RIGHT()`, and sort them.

Discussion

Suppose that a `housewares` table catalogs houseware furnishings, each identified by 10-character ID values consisting of three subparts: a three-character category abbreviation (such as `DIN` for "dining room" or `KIT` for "kitchen"), a five-digit serial number, and a two-character country code indicating where the part is manufactured:

```
mysql> SELECT * FROM housewares;
+------------+------------------+
| id         | description      |
+------------+------------------+
| DIN40672US | dining table     |
| KIT00372UK | garbage disposal |
| KIT01729JP | microwave oven   |
| BED00038SG | bedside lamp     |
| BTH00485US | shower stall     |
| BTH00415JP | lavatory         |
+------------+------------------+
```

This is not necessarily a good way to store complex ID values, and later we'll consider how to represent them using separate columns. For now, assume that the values must be stored as shown.

To sort rows from this table based on the `id` values, use the entire column value:

```
mysql> SELECT * FROM housewares ORDER BY id;
+------------+------------------+
| id         | description      |
+------------+------------------+
| BED00038SG | bedside lamp     |
| BTH00415JP | lavatory         |
| BTH00485US | shower stall     |
| DIN40672US | dining table     |
| KIT00372UK | garbage disposal |
| KIT01729JP | microwave oven   |
+------------+------------------+
```

But you might also have a need to sort on any of the three subparts (for example, to sort by country of manufacture). For that kind of operation, functions such as `LEFT()`, `MID()`, and `RIGHT()` are useful to extract `id` value components:

```
mysql> SELECT id,
    -> LEFT(id,3) AS category,
    -> MID(id,4,5) AS serial,
    -> RIGHT(id,2) AS country
    -> FROM housewares;
+------------+----------+--------+---------+
| id         | category | serial | country |
```

```
+------------+----------+--------+---------+
| DIN40672US | DIN      | 40672  | US      |
| KIT00372UK | KIT      | 00372  | UK      |
| KIT01729JP | KIT      | 01729  | JP      |
| BED00038SG | BED      | 00038  | SG      |
| BTH00485US | BTH      | 00485  | US      |
| BTH00415JP | BTH      | 00415  | JP      |
+------------+----------+--------+---------+
```

Those fixed-length substrings of the `id` values can be used for sorting, either alone or in combination. For example, to sort by product category, extract and use the category in the `ORDER BY` clause:

```
mysql> SELECT * FROM housewares ORDER BY LEFT(id,3);
+------------+------------------+
| id         | description      |
+------------+------------------+
| BED00038SG | bedside lamp     |
| BTH00485US | shower stall     |
| BTH00415JP | lavatory         |
| DIN40672US | dining table     |
| KIT00372UK | garbage disposal |
| KIT01729JP | microwave oven   |
+------------+------------------+
```

To sort by product serial number, use `MID()` to extract the middle five characters from the `id` values, beginning with the fourth:

```
mysql> SELECT * FROM housewares ORDER BY MID(id,4,5);
+------------+------------------+
| id         | description      |
+------------+------------------+
| BED00038SG | bedside lamp     |
| KIT00372UK | garbage disposal |
| BTH00415JP | lavatory         |
| BTH00485US | shower stall     |
| KIT01729JP | microwave oven   |
| DIN40672US | dining table     |
+------------+------------------+
```

This appears to be a numeric sort, but it's actually a string sort because `MID()` returns strings. The lexical and numeric sort order are the same in this case because the "numbers" have leading zeros to make them all the same length.

To sort by country code, use the rightmost two characters of the `id` values (`ORDER BY RIGHT(id,2)`).

You can also sort using combinations of substrings; for example, by country code and serial number within country:

```
mysql> SELECT * FROM housewares ORDER BY RIGHT(id,2), MID(id,4,5);
+------------+------------------+
| id         | description      |
```

```
+------------+------------------+
| BTH00415JP | lavatory         |
| KIT01729JP | microwave oven   |
| BED00038SG | bedside lamp     |
| KIT00372UK | garbage disposal |
| BTH00485US | shower stall     |
| DIN40672US | dining table     |
+------------+------------------+
```

The ORDER BY clauses just shown suffice to sort by substrings of the `id` values, but if such operations on the table are common, it might be worth representing houseware IDs differently; for example, using separate columns for the ID components. This table, `housewares2`, is like `housewares` but uses `category`, `serial`, and `country` columns rather than an `id` column:

```
CREATE TABLE housewares2
(
  category    VARCHAR(3) NOT NULL,
  serial      INT(5) UNSIGNED ZEROFILL NOT NULL,
  country     VARCHAR(2) NOT NULL,
  description VARCHAR(255),
  PRIMARY KEY (category, country, serial)
);
```

With the ID values split into separate parts, sorting operations are easier to specify; refer to individual columns directly rather than pulling out substrings of the original `id` column. You can also make operations that sort the `serial` and `country` columns more efficient by adding indexes on those columns. But a problem remains: how do you display each product ID as a single string rather than as three separate values? Do that with `CONCAT()`:

```
mysql> SELECT category, serial, country,
    -> CONCAT(category,serial,country) AS id
    -> FROM housewares2 ORDER BY category, country, serial;
+----------+--------+---------+------------+
| category | serial | country | id         |
+----------+--------+---------+------------+
| BED      |  00038 | SG      | BED00038SG |
| BTH      |  00415 | JP      | BTH00415JP |
| BTH      |  00485 | US      | BTH00485US |
| DIN      |  40672 | US      | DIN40672US |
| KIT      |  01729 | JP      | KIT01729JP |
| KIT      |  00372 | UK      | KIT00372UK |
+----------+--------+---------+------------+
```

This example illustrates an important principle: you might think about values one way (`id` values as single strings), but you need not necessarily represent them that way in the database. If an alternative representation (separate columns) is more efficient or easier to work with, it may well be worth using—even if you must reformat the underlying columns so they appear as people expect.

7.8. Sorting by Variable-Length Substrings

Problem

You want to sort using parts of a column that do *not* occur at a given position within the column.

Solution

Determine how to identify the parts you need so that you can extract them. Otherwise, you're out of luck.

Discussion

If substrings to be used for sorting vary in length, you need a reliable means of extracting just the part you want. To see how this works, create a `housewares3` table that is like the `housewares` table used in Recipe 7.7, except that it has no leading zeros in the serial number part of the `id` values:

```
mysql> SELECT * FROM housewares3;
+------------+------------------+
| id         | description      |
+------------+------------------+
| DIN40672US | dining table     |
| KIT372UK   | garbage disposal |
| KIT1729JP  | microwave oven   |
| BED38SG    | bedside lamp     |
| BTH485US   | shower stall     |
| BTH415JP   | lavatory         |
+------------+------------------+
```

The category and country parts of the `id` values can be extracted and sorted using `LEFT()` and `RIGHT()`, just as for the `housewares` table. But now the numeric segments of the values have different lengths and cannot be extracted and sorted using a simple `MID()` call. Instead, use `SUBSTRING()` to skip the first three characters. Of the remainder beginning with the fourth character (the first digit), take everything but the rightmost two columns. One way to do this is as follows:

```
mysql> SELECT id, LEFT(SUBSTRING(id,4),CHAR_LENGTH(SUBSTRING(id,4)-2))
    -> FROM housewares3;
+------------+-----------------------------------------------------+
| id         | LEFT(SUBSTRING(id,4),CHAR_LENGTH(SUBSTRING(id,4)-2)) |
+------------+-----------------------------------------------------+
| DIN40672US | 40672                                                |
| KIT372UK   | 372                                                  |
| KIT1729JP  | 1729                                                 |
| BED38SG    | 38                                                   |
| BTH485US   | 485                                                  |
```

```
| BTH415JP   | 415                                                  |
+------------+------------------------------------------------------+
```

But that's more complex than necessary. The `SUBSTRING()` function takes an optional third argument specifying a desired result length, and we know that the length of the middle part is equal to the length of the string minus five (three for the characters at the beginning and two for the characters at the end). The following query demonstrates how to get the numeric middle part by beginning with the ID, and then stripping the rightmost suffix:

```
mysql> SELECT id, SUBSTRING(id,4), SUBSTRING(id,4,CHAR_LENGTH(id)-5)
    -> FROM housewares3;
+------------+----------------+----------------------------------+
| id         | SUBSTRING(id,4) | SUBSTRING(id,4,CHAR_LENGTH(id)-5) |
+------------+----------------+----------------------------------+
| DIN40672US | 40672US         | 40672                            |
| KIT372UK   | 372UK           | 372                              |
| KIT1729JP  | 1729JP          | 1729                             |
| BED38SG    | 38SG            | 38                               |
| BTH485US   | 485US           | 485                              |
| BTH415JP   | 415JP           | 415                              |
+------------+----------------+----------------------------------+
```

Unfortunately, although the final expression correctly extracts the numeric part from the IDs, the resulting values are strings. Consequently, they sort lexically rather than numerically:

```
mysql> SELECT * FROM housewares3
    -> ORDER BY SUBSTRING(id,4,CHAR_LENGTH(id)-5);
+------------+------------------+
| id         | description      |
+------------+------------------+
| KIT1729JP  | microwave oven   |
| KIT372UK   | garbage disposal |
| BED38SG    | bedside lamp     |
| DIN40672US | dining table     |
| BTH415JP   | lavatory         |
| BTH485US   | shower stall     |
+------------+------------------+
```

How to deal with that? One way is to add zero, which tells MySQL to perform a string-to-number conversion that results in a numeric sort of the serial number values:

```
mysql> SELECT * FROM housewares3
    -> ORDER BY SUBSTRING(id,4,CHAR_LENGTH(id)-5)+0;
+------------+------------------+
| id         | description      |
+------------+------------------+
| BED38SG    | bedside lamp     |
| KIT372UK   | garbage disposal |
| BTH415JP   | lavatory         |
| BTH485US   | shower stall     |
```

```
| KIT1729JP  | microwave oven   |
| DIN40672US | dining table     |
+------------+------------------+
```

In this particular case, a simpler solution is possible. It's unnecessary to calculate the length of the numeric part of the string, because a string-to-number conversion operation strips trailing nonnumeric suffixes and provides the values needed to sort on the variable-length serial number portion of the `id` values. That means the third argument to `SUBSTRING()` actually isn't needed:

```
mysql> SELECT * FROM housewares3
    -> ORDER BY SUBSTRING(id,4)+0;
+------------+------------------+
| id         | description      |
+------------+------------------+
| BED38SG    | bedside lamp     |
| KIT372UK   | garbage disposal |
| BTH415JP   | lavatory         |
| BTH485US   | shower stall     |
| KIT1729JP  | microwave oven   |
| DIN40672US | dining table     |
+------------+------------------+
```

In the preceding example, the ability to extract variable-length substrings is based on the different kinds of characters in the middle of the `id` values, compared to the characters on the ends (that is, digits versus nondigits). In other cases, you may be able to use delimiter characters to pull apart column values. For the next examples, assume a `housewares4` table with `id` values that look like this:

```
mysql> SELECT * FROM housewares4;
+--------------+------------------+
| id           | description      |
+--------------+------------------+
| 13-478-92-2  | dining table     |
| 873-48-649-63 | garbage disposal |
| 8-4-2-1      | microwave oven   |
| 97-681-37-66 | bedside lamp     |
| 27-48-534-2  | shower stall     |
| 5764-56-89-72 | lavatory         |
+--------------+------------------+
```

To extract segments from these values, use `SUBSTRING_INDEX(`*`str`*`,`*`c`*`,`*`n`*`)`. It searches a string *`str`* for the *`n`*-th occurrence of a given character *`c`* and returns everything to the left of that character. For example, the following call returns `13-478`:

```
SUBSTRING_INDEX('13-478-92-2','-',2)
```

If *`n`* is negative, the search for *`c`* proceeds from the right and returns the rightmost string. This call returns `478-92-2`:

```
SUBSTRING_INDEX('13-478-92-2','-',-3)
```

By combining SUBSTRING_INDEX() calls with positive and negative indexes, it's possible to extract successive pieces from each id value: extract the first *n* segments of the value and pull off the rightmost one. By varying *n* from 1 to 4, we get the successive segments from left to right:

```
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',1),'-',-1)
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',2),'-',-1)
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',3),'-',-1)
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',4),'-',-1)
```

The first of those expressions can be optimized because the inner SUBSTRING_INDEX() call returns a single-segment string and is sufficient by itself to return the leftmost id segment:

```
SUBSTRING_INDEX(id,'-',1)
```

Another way to obtain substrings is to extract the rightmost *n* segments of the value and pull off the first one. Here we vary *n* from –4 to –1:

```
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',-4),'-',1)
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',-3),'-',1)
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',-2),'-',1)
SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',-1),'-',1)
```

Again, an optimization is possible. For the fourth expression, the inner SUBSTRING_INDEX() call is sufficient to return the final substring:

```
SUBSTRING_INDEX(id,'-',-1)
```

These expressions can be difficult to read and understand, and experimenting with a few to see how they work may be useful. Here is an example that shows how to get the second and fourth segments from the id values:

```
mysql> SELECT
    -> id,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',2),'-',-1) AS segment2,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',4),'-',-1) AS segment4
    -> FROM housewares4;
+--------------+----------+----------+
| id           | segment2 | segment4 |
+--------------+----------+----------+
| 13-478-92-2  | 478      | 2        |
| 873-48-649-63 | 48      | 63       |
| 8-4-2-1      | 4        | 1        |
| 97-681-37-66 | 681      | 66       |
| 27-48-534-2  | 48       | 2        |
| 5764-56-89-72 | 56      | 72       |
+--------------+----------+----------+
```

To use the substrings for sorting, use the appropriate expressions in the ORDER BY clause. (Remember to force a string-to-number conversion by adding zero if you want a

numeric rather than lexical sort.) The following two queries order the results based on the second `id` segment. The first sorts lexically, the second numerically:

```
mysql> SELECT * FROM housewares4
    -> ORDER BY SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',2),'-',-1);
+---------------+------------------+
| id            | description      |
+---------------+------------------+
| 8-4-2-1       | microwave oven   |
| 13-478-92-2   | dining table     |
| 873-48-649-63 | garbage disposal |
| 27-48-534-2   | shower stall     |
| 5764-56-89-72 | lavatory         |
| 97-681-37-66  | bedside lamp     |
+---------------+------------------+
mysql> SELECT * FROM housewares4
    -> ORDER BY SUBSTRING_INDEX(SUBSTRING_INDEX(id,'-',2),'-',-1)+0;
+---------------+------------------+
| id            | description      |
+---------------+------------------+
| 8-4-2-1       | microwave oven   |
| 873-48-649-63 | garbage disposal |
| 27-48-534-2   | shower stall     |
| 5764-56-89-72 | lavatory         |
| 13-478-92-2   | dining table     |
| 97-681-37-66  | bedside lamp     |
+---------------+------------------+
```

The substring-extraction expressions here are messy, but at least the column values to which we apply the expressions have a consistent number of segments. To sort values that have varying numbers of segments, the job can be more difficult. Recipe 7.9 shows an example illustrating why that is.

7.9. Sorting Hostnames in Domain Order

Problem

You want to sort hostnames in domain order, with the rightmost parts of the names more significant than the leftmost parts.

Solution

Break apart the names, and sort the pieces from right to left.

Discussion

Hostnames are strings and therefore their natural sort order is lexical. However, it's often desirable to sort hostnames in domain order, where the rightmost segments of the

hostname values are more significant than the leftmost segments. Suppose that a `hostname` table contains the following names:

```
mysql> SELECT name FROM hostname ORDER BY name;
+--------------------+
| name               |
+--------------------+
| dbi.perl.org       |
| jakarta.apache.org |
| lists.mysql.com    |
| mysql.com          |
| svn.php.net        |
| www.kitebird.com   |
+--------------------+
```

The preceding query demonstrates the natural lexical sort order of the `name` values. That differs from domain order, as the following table shows:

Lexical order	Domain order
dbi.perl.org	www.kitebird.com
jakarta.apache.org	mysql.com
lists.mysql.com	lists.mysql.com
mysql.com	svn.php.net
svn.php.net	jakarta.apache.org
www.kitebird.com	dbi.perl.org

Producing domain-ordered output is a substring-sorting problem for which it's necessary to extract each segment of the names so they can be sorted in right-to-left fashion. There is also an additional complication if your values contain different numbers of segments, as our example hostnames do. (Most of them have three segments, but `mysql.com` has only two.)

To extract the pieces of the hostnames, begin by using `SUBSTRING_INDEX()` in a manner similar to that described previously in Recipe 7.8. The hostname values have a maximum of three segments, from which the pieces can be extracted left to right like this:

```
SUBSTRING_INDEX(SUBSTRING_INDEX(name,'.',-3),'.',1)
SUBSTRING_INDEX(SUBSTRING_INDEX(name,'.',-2),'.',1)
SUBSTRING_INDEX(name,'.',-1)
```

These expressions work properly as long as all the hostnames have three components. But if a name has fewer than three, you don't get the correct result, as the following query demonstrates:

```
mysql> SELECT name,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(name,'.',-3),'.',1) AS leftmost,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(name,'.',-2),'.',1) AS middle,
    -> SUBSTRING_INDEX(name,'.',-1) AS rightmost
    -> FROM hostname;
```

```
+--------------------+----------+----------+-----------+
| name               | leftmost | middle   | rightmost |
+--------------------+----------+----------+-----------+
| svn.php.net        | svn      | php      | net       |
| dbi.perl.org       | dbi      | perl     | org       |
| lists.mysql.com    | lists    | mysql    | com       |
| mysql.com          | mysql    | mysql    | com       |
| jakarta.apache.org | jakarta  | apache   | org       |
| www.kitebird.com   | www      | kitebird | com       |
+--------------------+----------+----------+-----------+
```

Notice the output for the `mysql.com` row; it has `mysql` for the value of the `leftmost` column, where it should have an empty string. The segment-extraction expressions work by pulling off the rightmost *n* segments, and then returning the leftmost segment of the result. The source of the problem for `mysql.com` is that if there aren't *n* segments, the expression simply returns the leftmost segment of however many there are. To fix this problem, add a sufficient number of periods at the beginning of the hostname values to guarantee that they have the requisite number of segments:

```
mysql> SELECT name,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(CONCAT('..',name),'.',-3),'.',1)
    -> AS leftmost,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(CONCAT('.',name),'.',-2),'.',1)
    -> AS middle,
    -> SUBSTRING_INDEX(name,'.',-1) AS rightmost
    -> FROM hostname;
+--------------------+----------+----------+-----------+
| name               | leftmost | middle   | rightmost |
+--------------------+----------+----------+-----------+
| svn.php.net        | svn      | php      | net       |
| dbi.perl.org       | dbi      | perl     | org       |
| lists.mysql.com    | lists    | mysql    | com       |
| mysql.com          |          | mysql    | com       |
| jakarta.apache.org | jakarta  | apache   | org       |
| www.kitebird.com   | www      | kitebird | com       |
+--------------------+----------+----------+-----------+
```

That's pretty ugly. But the expressions do serve to extract the substrings that are needed for sorting hostname values correctly in right-to-left fashion:

```
mysql> SELECT name FROM hostname
    -> ORDER BY
    -> SUBSTRING_INDEX(name,'.',-1),
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(CONCAT('.',name),'.',-2),'.',1),
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(CONCAT('..',name),'.',-3),'.',1);
+--------------------+
| name               |
+--------------------+
| www.kitebird.com   |
| mysql.com          |
| lists.mysql.com    |
| svn.php.net        |
```

```
| jakarta.apache.org |
| dbi.perl.org       |
+--------------------+
```

If your hostnames have a maximum of four segments rather than three, add to the `ORDER BY` clause another `SUBSTRING_INDEX()` expression that adds three dots at the beginning of the hostname values.

7.10. Sorting Dotted-Quad IP Values in Numeric Order

Problem

You want to sort in numeric order strings that represent IP numbers.

Solution

Break apart the strings, and sort the pieces numerically. Or just use `INET_ATON()`. Or consider storing the values as numbers instead.

Discussion

If a table contains IP numbers represented as strings in dotted-quad notation (`192.168.1.10`), they sort lexically rather than numerically. To produce a numeric ordering instead, sort them as four-part values with each part sorted numerically. Or, to be more efficient, represent the IP numbers as 32-bit unsigned integers, which take less space and can be ordered by a simple numeric sort. This section shows both methods.

To sort string-valued dotted-quad IP numbers, use a technique similar to that for sorting hostnames (see Recipe 7.9), but with the following differences:

- Dotted quads always have four segments. There's no need to add dots to the value before extracting substrings.
- Dotted quads sort left to right. The order of the substrings used in the `ORDER BY` clause is opposite to that used for hostname sorting.
- The segments of dotted-quad values are numbers. Add zero to each substring to force a numeric rather than lexical sort.

Suppose that a `hostip` table has a string-valued `ip` column containing IP numbers:

```
mysql> SELECT ip FROM hostip ORDER BY ip;
+-----------------+
| ip              |
+-----------------+
| 127.0.0.1       |
| 192.168.0.10    |
| 192.168.0.2     |
```

```
| 192.168.1.10    |
| 192.168.1.2     |
| 21.0.0.1        |
| 255.255.255.255 |
+-----------------+
```

The preceding query produces output sorted in lexical order. To sort the `ip` values numerically, extract each segment and add zero to convert it to a number like this:

```
mysql> SELECT ip FROM hostip
    -> ORDER BY
    -> SUBSTRING_INDEX(ip,'.',1)+0,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(ip,'.',-3),'.',1)+0,
    -> SUBSTRING_INDEX(SUBSTRING_INDEX(ip,'.',-2),'.',1)+0,
    -> SUBSTRING_INDEX(ip,'.',-1)+0;
+-----------------+
| ip              |
+-----------------+
| 21.0.0.1        |
| 127.0.0.1       |
| 192.168.0.2     |
| 192.168.0.10    |
| 192.168.1.2     |
| 192.168.1.10    |
| 255.255.255.255 |
+-----------------+
```

However, although that `ORDER BY` clause produces a correct result, it's complicated. A simpler solution uses the `INET_ATON()` function to convert network addresses in string form to their underlying numeric values, then sorts those numbers:

```
mysql> SELECT ip FROM hostip ORDER BY INET_ATON(ip);
+-----------------+
| ip              |
+-----------------+
| 21.0.0.1        |
| 127.0.0.1       |
| 192.168.0.2     |
| 192.168.0.10    |
| 192.168.1.2     |
| 192.168.1.10    |
| 255.255.255.255 |
+-----------------+
```

If you're tempted to sort by simply adding zero to the `ip` value and using `ORDER BY` on the result, consider the values that kind of string-to-number conversion actually produces:

```
mysql> SELECT ip, ip+0 FROM hostip;
+-----------------+---------+
| ip              | ip+0    |
+-----------------+---------+
| 127.0.0.1       |     127 |
```

```
| 192.168.0.2     | 192.168 |
| 192.168.0.10    | 192.168 |
| 192.168.1.2     | 192.168 |
| 192.168.1.10    | 192.168 |
| 255.255.255.255 | 255.255 |
| 21.0.0.1        |      21 |
+-----------------+---------+
7 rows in set, 7 warnings (0.00 sec)
```

The conversion retains only as much of each value as can be interpreted as a valid number (hence the warnings). The remainder becomes unavailable for sorting purposes, even though it's required for a correct ordering.

Use of `INET_ATON()` in the `ORDER BY` clause is more efficient than six `SUBSTRING_INDEX()` calls. Moreover, if you're willing to consider storing IP addresses as numbers rather than as strings, you can avoid performing any conversion at all when sorting. You gain other benefits as well: numeric IP addresses have 32 bits, so you can use a 4-byte `INT UNSIGNED` column to store them, which requires less storage than the string form. Also, if you index the column, the query optimizer may be able to use the index for certain queries. For cases requiring display of numeric IP values in dotted-quad notation, convert them with the `INET_NTOA()` function.

7.11. Floating Values to the Head or Tail of the Sort Order

Problem

You want a column to sort the way it normally does, except for a few values that should appear at the beginning or end of the sort order. For example, you want to sort a list in lexical order except for certain high-priority values that should appear first no matter where they fall in the normal sort order.

Solution

Add an initial sort column to the `ORDER BY` clause that places those few values where you want them. The remaining sort columns have their usual effect for the other values.

Discussion

To sort a result set normally *except* that you want particular values first, create an additional sort column that is 0 for those values and 1 for everything else. This enables you to float the values to the head of the sort order. To put the values at the tail instead, use the additional column to map the values to 1 and all other values to 0.

Suppose that a column contains NULL values:

```
mysql> SELECT val FROM t;
+------+
| val  |
+------+
|    3 |
|  100 |
| NULL |
| NULL |
|    9 |
+------+
```

Normally, sorting groups the NULL values at the beginning for an ascending sort:

```
mysql> SELECT val FROM t ORDER BY val;
+------+
| val  |
+------+
| NULL |
| NULL |
|    3 |
|    9 |
|  100 |
+------+
```

To put them at the end instead, without changing the order of other values, introduce an extra ORDER BY column that maps NULL values to a higher value than non-NULL values:

```
mysql> SELECT val FROM t ORDER BY IF(val IS NULL,1,0), val;
+------+
| val  |
+------+
|    3 |
|    9 |
|  100 |
| NULL |
| NULL |
+------+
```

The IF() expression creates a new column for the sort that is used as the primary sort value.

For descending sorts, NULL values group at the end. To put them at the beginning instead, use the same technique, but reverse the second and third arguments of the IF() function to map NULL values to a lower value than non-NULL values:

```
IF(val IS NULL,0,1)
```

The same technique is useful for floating values other than NULL to either end of the sort order. Suppose that you want to sort mail table messages in sender/recipient order, but you want to put messages for a particular sender first. In the real world, the most

interesting sender might be `postmaster` or `root`. Those names don't appear in the table, so let's use `phil` as the name of interest instead:

```
mysql> SELECT t, srcuser, dstuser, size
    -> FROM mail
    -> ORDER BY IF(srcuser='phil',0,1), srcuser, dstuser;
+---------------------+---------+---------+---------+
| t                   | srcuser | dstuser | size    |
+---------------------+---------+---------+---------+
| 2014-05-16 23:04:19 | phil    | barb    |   10294 |
| 2014-05-12 15:02:49 | phil    | phil    |    1048 |
| 2014-05-15 08:50:57 | phil    | phil    |     978 |
| 2014-05-14 11:52:17 | phil    | tricia  |    5781 |
| 2014-05-19 12:49:23 | phil    | tricia  |     873 |
| 2014-05-14 14:42:21 | barb    | barb    |   98151 |
| 2014-05-11 10:15:08 | barb    | tricia  |   58274 |
| 2014-05-12 18:59:18 | barb    | tricia  |     271 |
| 2014-05-14 09:31:37 | gene    | barb    |    2291 |
| 2014-05-16 09:00:28 | gene    | barb    |     613 |
| 2014-05-15 17:35:31 | gene    | gene    |    3856 |
| 2014-05-15 07:17:48 | gene    | gene    |    3824 |
| 2014-05-19 22:21:51 | gene    | gene    |   23992 |
| 2014-05-15 10:25:52 | gene    | tricia  |  998532 |
| 2014-05-12 12:48:13 | tricia  | gene    |  194925 |
| 2014-05-14 17:03:01 | tricia  | phil    | 2394482 |
+---------------------+---------+---------+---------+
```

The value of the extra sort column is `0` for rows in which the `srcuser` value is `phil`, and `1` for all other rows. By making that the most significant sort column, rows for messages sent by `phil` float to the top of the output. (To sink them to the bottom instead, either sort the column in reverse order using `DESC`, or reverse the order of the second and third arguments of the `IF()` function.)

You can also use this technique for particular conditions, not only specific values. To put first those rows where people sent messages to themselves, do this:

```
mysql> SELECT t, srcuser, dstuser, size
    -> FROM mail
    -> ORDER BY IF(srcuser=dstuser,0,1), srcuser, dstuser;
+---------------------+---------+---------+---------+
| t                   | srcuser | dstuser | size    |
+---------------------+---------+---------+---------+
| 2014-05-14 14:42:21 | barb    | barb    |   98151 |
| 2014-05-19 22:21:51 | gene    | gene    |   23992 |
| 2014-05-15 17:35:31 | gene    | gene    |    3856 |
| 2014-05-15 07:17:48 | gene    | gene    |    3824 |
| 2014-05-12 15:02:49 | phil    | phil    |    1048 |
| 2014-05-15 08:50:57 | phil    | phil    |     978 |
| 2014-05-11 10:15:08 | barb    | tricia  |   58274 |
| 2014-05-12 18:59:18 | barb    | tricia  |     271 |
| 2014-05-16 09:00:28 | gene    | barb    |     613 |
| 2014-05-14 09:31:37 | gene    | barb    |    2291 |
```

```
| 2014-05-15 10:25:52 | gene    | tricia  |  998532 |
| 2014-05-16 23:04:19 | phil    | barb    |   10294 |
| 2014-05-14 11:52:17 | phil    | tricia  |    5781 |
| 2014-05-19 12:49:23 | phil    | tricia  |     873 |
| 2014-05-12 12:48:13 | tricia  | gene    |  194925 |
| 2014-05-14 17:03:01 | tricia  | phil    | 2394482 |
+---------------------+---------+---------+---------+
```

If you have a pretty good idea about the contents of your table, it's sometimes possible to eliminate the extra sort column. For example, `srcuser` is never `NULL` in the `mail` table, so the previous query can be rewritten as follows to use one less column in the `ORDER BY` clause (this relies on the property that `NULL` values sort ahead of all non-`NULL` values):

```
SELECT t, srcuser, dstuser, size
FROM mail
ORDER BY IF(srcuser=dstuser,NULL,srcuser), dstuser;
```

7.12. Defining a Custom Sort Order

Problem

You want to sort values in a nonstandard order.

Solution

Use `FIELD()` to map column values to a sequence that places the values in the desired order.

Discussion

Recipe 7.11 shows how to make a specific group of rows float to the head of the sort order. To impose a specific order on *all* values in a column, use the `FIELD()` function to map them to a list of numeric values and use the numbers for sorting. `FIELD()` compares its first argument to the following arguments and returns an integer indicating which one it matches. (This works best when the column contains a small number of distinct values.)

The following `FIELD()` call compares *`value`* to *`str1`*, *`str2`*, *`str3`*, and *`str4`*, and returns 1, 2, 3, or 4, depending on which of them *`value`* is equal to:

```
FIELD(value,str1,str2,str3,str4)
```

If *`value`* is `NULL` or none of the values match, `FIELD()` returns 0.

You can use `FIELD()` to sort an arbitrary set of values into any order you please. For example, to display `driver_log` rows for Henry, Suzi, and Ben, in that order, do this:

```
mysql> SELECT * FROM driver_log
    -> ORDER BY FIELD(name,'Henry','Suzi','Ben');
```

```
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|     10 | Henry | 2014-07-30 |   203 |
|      8 | Henry | 2014-08-01 |   197 |
|      6 | Henry | 2014-07-26 |   115 |
|      4 | Henry | 2014-07-27 |    96 |
|      3 | Henry | 2014-07-29 |   300 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      2 | Suzi  | 2014-07-29 |   391 |
|      5 | Ben   | 2014-07-29 |   131 |
|      9 | Ben   | 2014-08-02 |    79 |
|      1 | Ben   | 2014-07-30 |   152 |
+--------+-------+------------+-------+
```

7.13. Sorting ENUM Values

Problem

`ENUM` values don't sort like other string columns.

Solution

Learn how they work, and exploit those properties to your advantage.

Discussion

`ENUM` is a string data type, but `ENUM` values actually are stored numerically with values ordered the same way they are listed in the table definition. These numeric values affect how enumerations are sorted, which can be very useful. Suppose that a table named `weekday` contains an enumeration column named `day` that has weekday names as its members:

```
CREATE TABLE weekday
(
  day ENUM('Sunday','Monday','Tuesday','Wednesday',
           'Thursday','Friday','Saturday')
);
```

Internally, MySQL defines the enumeration values `Sunday` through `Saturday` in that definition to have numeric values from 1 to 7. To see this for yourself, create the table using the definition just shown, and then insert into it a row for each day of the week. To make the insertion order differ from sorted order (so that you can see the effect of sorting), add the days in random order:

```
mysql> INSERT INTO weekday (day) VALUES('Monday'),('Friday'),
    -> ('Tuesday'), ('Sunday'), ('Thursday'), ('Saturday'), ('Wednesday');
```

Then select the values, both as strings and as the internal numeric value (obtain the latter using `+0` to force a string-to-number conversion):

```
mysql> SELECT day, day+0 FROM weekday;
+-----------+-------+
| day       | day+0 |
+-----------+-------+
| Monday    |     2 |
| Friday    |     6 |
| Tuesday   |     3 |
| Sunday    |     1 |
| Thursday  |     5 |
| Saturday  |     7 |
| Wednesday |     4 |
+-----------+-------+
```

Notice that because the query includes no `ORDER BY` clause, the rows are returned in unsorted order. If you add an `ORDER BY day` clause, it becomes apparent that MySQL uses the internal numeric values for sorting:

```
mysql> SELECT day, day+0 FROM weekday ORDER BY day;
+-----------+-------+
| day       | day+0 |
+-----------+-------+
| Sunday    |     1 |
| Monday    |     2 |
| Tuesday   |     3 |
| Wednesday |     4 |
| Thursday  |     5 |
| Friday    |     6 |
| Saturday  |     7 |
+-----------+-------+
```

What about occasions when you want to sort `ENUM` values in lexical order? Force them to be treated as strings for sorting using the `CAST()` function:

```
mysql> SELECT day, day+0 FROM weekday ORDER BY CAST(day AS CHAR);
+-----------+-------+
| day       | day+0 |
+-----------+-------+
| Friday    |     6 |
| Monday    |     2 |
| Saturday  |     7 |
| Sunday    |     1 |
| Thursday  |     5 |
| Tuesday   |     3 |
| Wednesday |     4 |
+-----------+-------+
```

If you always (or nearly always) sort a non-enumeration column in a specific nonlexical order, consider changing the data type to `ENUM`, with its values listed in the desired sort

order. To see how this works, create a `color` table containing a string column, and populate it with some sample rows:

```
mysql> CREATE TABLE color (name CHAR(10));
mysql> INSERT INTO color (name) VALUES ('blue'),('green'),
    -> ('indigo'),('orange'),('red'),('violet'),('yellow');
```

Sorting by the `name` column at this point produces lexical order because the column contains `CHAR` values:

```
mysql> SELECT name FROM color ORDER BY name;
+--------+
| name   |
+--------+
| blue   |
| green  |
| indigo |
| orange |
| red    |
| violet |
| yellow |
+--------+
```

Now suppose that you want to sort the column by the order in which colors occur in the rainbow. (This is "Roy G. Biv" order; successive letters of that name indicate the first letters of the corresponding color names.) One way to produce a rainbow sort is to use `FIELD()`:

```
mysql> SELECT name FROM color
    -> ORDER BY
    -> FIELD(name,'red','orange','yellow','green','blue','indigo','violet');
+--------+
| name   |
+--------+
| red    |
| orange |
| yellow |
| green  |
| blue   |
| indigo |
| violet |
+--------+
```

To accomplish the same end without `FIELD()`, use `ALTER TABLE` to convert the `name` column to an `ENUM` that lists the colors in the desired sort order:

```
mysql> ALTER TABLE color
    -> MODIFY name
    -> ENUM('red','orange','yellow','green','blue','indigo','violet');
```

After converting the table, sorting on the `name` column produces rainbow sorting naturally with no special treatment:

```
mysql> SELECT name FROM color ORDER BY name;
+--------+
| name   |
+--------+
| red    |
| orange |
| yellow |
| green  |
| blue   |
| indigo |
| violet |
+--------+
```

CHAPTER 8
Generating Summaries

8.0. Introduction

Database systems are useful for data storage and retrieval, but can also summarize your data in more concise forms. Summaries are useful when you want the overall picture, not the details. They're more readily understood than a long list of records. They enable you to answer questions such as "How many?" or "What is the total?" or "What is the range of values?" If you run a business, you may want to know how many customers you have in each state, or how much sales volume you generate each month.

The preceding examples include two common summary types: counting summaries and content summaries. The first (the number of customer records per state) is a counting summary. The content of each record is important only for purposes of placing it into the proper group or category for counting. Such summaries are essentially histograms, where you sort items into a set of bins and count the number of items in each bin. The second example (sales volume per month) is a content summary, in which sales totals are based on sales values in order records.

Another summary type produces neither counts nor sums, but simply a list of unique values. This is useful if you care *which* values are present rather than how many of each there are. To determine the states in which you have customers, you need a list of the distinct state names contained in the records, not a list consisting of the state value from every record.

The summary types available to you depend on the nature of your data. A counting summary can be generated from all kinds of values, whether they be numbers, strings, or dates. Summaries that produce sums or averages apply only to numeric values. You can count instances of customer state names to produce a demographic analysis of your customer base. And sometimes it makes sense to apply one summary technique to the result of another. For example, to determine how many states your customers live in, generate a list of unique customer states, then count them.

Summary operations in MySQL involve the following SQL constructs:

- To compute a summary value from a set of individual values, use one of the functions known as aggregate functions. These are so called because they operate on aggregates (groups) of values. Aggregate functions include `COUNT()`, which counts rows or values in a query result; `MIN()` and `MAX()`, which find smallest and largest values; and `SUM()` and `AVG()`, which produce sums and means of values. These functions can be used to compute a value for the entire result set, or with a `GROUP BY` clause to group rows into subsets and obtain an aggregate value for each one.
- To obtain a list of unique values, use `SELECT DISTINCT` rather than `SELECT`.
- To count unique values, use `COUNT(DISTINCT)` rather than `COUNT()`.

The recipes in this chapter first illustrate basic summary techniques, and then show how to perform more complex summary operations. You'll find additional examples of summary methods in later chapters, particularly those that cover joins and statistical operations. (See Chapter 14 and Chapter 15.)

Summary queries sometimes involve complex expressions. For summaries that you execute often, keep in mind that views can make queries easier to use. Recipe 3.7 demonstrates the basic technique of creating a view. Recipe 8.2 shows how it applies to summary simplification, and you'll easily see how it can be used in later sections of the chapter as well.

The primary tables used for examples in this chapter are the `driver_log` and `mail` tables. These were also used in Chapter 7, so they should look familiar. A third table used throughout the chapter is `states`, which has rows containing a few columns of information for each of the United States:

```
mysql> SELECT * FROM states ORDER BY name;
+----------------+--------+------------+----------+
| name           | abbrev | statehood  | pop      |
+----------------+--------+------------+----------+
| Alabama        | AL     | 1819-12-14 |  4779736 |
| Alaska         | AK     | 1959-01-03 |   710231 |
| Arizona        | AZ     | 1912-02-14 |  6392017 |
| Arkansas       | AR     | 1836-06-15 |  2915918 |
| California     | CA     | 1850-09-09 | 37253956 |
| Colorado       | CO     | 1876-08-01 |  5029196 |
| Connecticut    | CT     | 1788-01-09 |  3574097 |
…
```

The `name` and `abbrev` columns list the full state name and the corresponding abbreviation. The `statehood` column indicates the day on which the state entered the Union. `pop` is the state population from the 2010 census, as reported by the US Census Bureau.

This chapter uses other tables occasionally as well. You can create them with scripts found in the *tables* directory of the `recipes` distribution. Recipe 5.12 describes the `kjv` table.

8.1. Basic Summary Techniques

Problem

You want to summarize a dataset in various ways, such as counting the number of rows that match certain conditions, determining the smallest or largest of a set of values, adding or averaging a set of numbers, or finding which unique values are present.

Solution

Use the appropriate aggregate function to summarize values, `DISTINCT` to select unique values, or `COUNT(DISTINCT)` to count unique values.

Discussion

The following discussion illustrates how to apply the aggregate functions to produce basic summaries, and how to use `DISTINCT` to find unique values.

Summarizing with COUNT()

To count the number of rows in an entire table or that match particular conditions, use the `COUNT()` function. For example, to display the rows in a table, use a `SELECT *` statement, but to count them instead, use `SELECT COUNT(*)`. Without a `WHERE` clause, the statement counts all the rows in the table, such as in the following statement that shows how many rows the `driver_log` table contains:

```
mysql> SELECT COUNT(*) FROM driver_log;
+----------+
| COUNT(*) |
+----------+
|       10 |
+----------+
```

If you don't know how many US states there are (perhaps you think there are 57?), this statement tells you:

```
mysql> SELECT COUNT(*) FROM states;
+----------+
| COUNT(*) |
+----------+
|       50 |
+----------+
```

`COUNT(*)` with no `WHERE` clause performs a full table scan. For MyISAM tables, this is very quick. For InnoDB tables, you may want to avoid it because it can be slow for large tables. If an approximate row count is good enough, avoid a full scan by extracting the `TABLE_ROWS` value from the `INFORMATION_SCHEMA` database:

```
SELECT TABLE_ROWS FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'states';
```

To count only the number of rows that match certain conditions, include an appropriate `WHERE` clause in a `SELECT COUNT(*)` statement. The conditions can be chosen to make `COUNT(*)` useful for answering many kinds of questions:

- How many times did drivers travel more than 200 miles in a day?

```
mysql> SELECT COUNT(*) FROM driver_log WHERE miles > 200;
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

- How many days did Suzi drive?

```
mysql> SELECT COUNT(*) FROM driver_log WHERE name = 'Suzi';
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

- How many of the United States joined the Union in the 19th century?

```
mysql> SELECT COUNT(*) FROM states
    -> WHERE statehood BETWEEN '1800-01-01' AND '1899-12-31';
+----------+
| COUNT(*) |
+----------+
|       29 |
+----------+
```

The `COUNT()` function actually has two forms. The form we've been using, `COUNT(*)`, counts rows. The other form, `COUNT(`*`expr`*`)`, takes a column name or expression argument and counts the number of non-`NULL` values. The following statement shows how to produce both a row count for a table and a count of the number of non-`NULL` values in one of its columns:

```
SELECT COUNT(*), COUNT(mycol) FROM mytbl;
```

The fact that `COUNT(`*`expr`*`)` doesn't count `NULL` values is useful for producing multiple counts from the same set of rows. To count the number of Saturday and Sunday trips in the `driver_log` table with a single statement, do this:

```
mysql> SELECT
    -> COUNT(IF(DAYOFWEEK(trav_date)=7,1,NULL)) AS 'Saturday trips',
    -> COUNT(IF(DAYOFWEEK(trav_date)=1,1,NULL)) AS 'Sunday trips'
    -> FROM driver_log;
+---------------+-------------+
| Saturday trips | Sunday trips |
+---------------+-------------+
|              3 |           1 |
+---------------+-------------+
```

Or to count weekend versus weekday trips, do this:

```
mysql> SELECT
    -> COUNT(IF(DAYOFWEEK(trav_date) IN (1,7),1,NULL)) AS 'weekend trips',
    -> COUNT(IF(DAYOFWEEK(trav_date) IN (1,7),NULL,1)) AS 'weekday trips'
    -> FROM driver_log;
+---------------+---------------+
| weekend trips | weekday trips |
+---------------+---------------+
|             4 |             6 |
+---------------+---------------+
```

The `IF()` expressions determine, for each column value, whether it should be counted. If so, the expression evaluates to `1` and `COUNT()` counts it. If not, the expression evaluates to `NULL` and `COUNT()` ignores it. The effect is to count the number of values that satisfy the condition given as the first argument to `IF()`.

Summarizing with MIN() and MAX()

Finding smallest or largest values in a dataset is somewhat akin to sorting, except that instead of producing an entire set of sorted values, you select only a single value at one end or the other of the sorted range. This operation applies to questions about smallest, largest, oldest, newest, most expensive, least expensive, and so forth. One way to find such values is to use the `MIN()` and `MAX()` functions. (Another way is to use `LIMIT`; see Recipe 3.9.)

Because `MIN()` and `MAX()` determine the extreme values in a set, they're useful for characterizing ranges:

- What date range is represented by the rows in the `mail` table? What are the smallest and largest messages sent?

```
mysql> SELECT
    -> MIN(t) AS earliest, MAX(t) AS latest,
    -> MIN(size) AS smallest, MAX(size) AS largest
    -> FROM mail;
+---------------------+---------------------+----------+---------+
| earliest            | latest              | smallest | largest |
+---------------------+---------------------+----------+---------+
| 2014-05-11 10:15:08 | 2014-05-19 22:21:51 |      271 | 2394482 |
+---------------------+---------------------+----------+---------+
```

- What are the smallest and largest US state populations?

```
mysql> SELECT MIN(pop) AS 'fewest people', MAX(pop) AS 'most people'
    -> FROM states;
+---------------+-------------+
| fewest people | most people |
+---------------+-------------+
|        563626 |    37253956 |
+---------------+-------------+
```

- What are the first and last state names, lexically speaking? The shortest and longest names?

```
mysql> SELECT
    -> MIN(name) AS first,
    -> MAX(name) AS last,
    -> MIN(CHAR_LENGTH(name)) AS shortest,
    -> MAX(CHAR_LENGTH(name)) AS longest
    -> FROM states;
+---------+---------+----------+---------+
| first   | last    | shortest | longest |
+---------+---------+----------+---------+
| Alabama | Wyoming |        4 |      14 |
+---------+---------+----------+---------+
```

The final query illustrates that `MIN()` and `MAX()` need not be applied directly to column values; they're also useful for expressions or values derived from column values.

Summarizing with SUM() and AVG()

`SUM()` and `AVG()` produce the total and average (mean) of a set of values:

- What is the total amount of mail traffic in bytes and the average size of each message?

```
mysql> SELECT
    -> SUM(size) AS 'total traffic',
    -> AVG(size) AS 'average message size'
    -> FROM mail;
+---------------+----------------------+
| total traffic | average message size |
+---------------+----------------------+
|       3798185 |          237386.5625 |
+---------------+----------------------+
```

- How many miles did the drivers in the `driver_log` table travel? What was the average number of miles traveled per day?

```
mysql> SELECT
    -> SUM(miles) AS 'total miles',
    -> AVG(miles) AS 'average miles/day'
    -> FROM driver_log;
+-------------+-------------------+
| total miles | average miles/day |
```

```
+-------------+------------------+
|        2166 |          216.6000 |
+-------------+------------------+
```

- What is the total population of the United States?

```
mysql> SELECT SUM(pop) FROM states;
+-----------+
| SUM(pop)  |
+-----------+
| 308143815 |
+-----------+
```

  The value represents the population reported for the 2010 census. The figure shown here differs from the US population reported by the US Census Bureau because the `states` table contains no count for Washington, D.C.

`SUM()` and `AVG()` are numeric functions, so they can't be used with strings or temporal values. But sometimes you can convert nonnumeric values to useful numeric forms. Suppose that a table stores `TIME` values that represent elapsed time:

```
mysql> SELECT t1 FROM time_val;
+----------+
| t1       |
+----------+
| 15:00:00 |
| 05:01:30 |
| 12:30:20 |
+----------+
```

To compute the total elapsed time, use `TIME_TO_SEC()` to convert the values to seconds before summing them. The resulting sum is also in seconds; pass it to `SEC_TO_TIME()` to convert it back to `TIME` format:

```
mysql> SELECT SUM(TIME_TO_SEC(t1)) AS 'total seconds',
    -> SEC_TO_TIME(SUM(TIME_TO_SEC(t1))) AS 'total time'
    -> FROM time_val;
+---------------+------------+
| total seconds | total time |
+---------------+------------+
|        117110 | 32:31:50   |
+---------------+------------+
```

Using DISTINCT to eliminate duplicates

A summary operation that uses no aggregate functions is determining the unique values or rows in a dataset. Do this with `DISTINCT` (or `DISTINCTROW`, a synonym). `DISTINCT` boils down a query result, and often is combined with `ORDER BY` to place values in more meaningful order. This query lists in lexical order the drivers named in the `driver_log` table:

```
mysql> SELECT DISTINCT name FROM driver_log ORDER BY name;
+-------+
| name  |
+-------+
| Ben   |
| Henry |
| Suzi  |
+-------+
```

Without `DISTINCT`, the statement produces the same names, but is not nearly as easy to understand, even with a small dataset:

```
mysql> SELECT name FROM driver_log ORDER BY NAME;
+-------+
| name  |
+-------+
| Ben   |
| Ben   |
| Ben   |
| Henry |
| Henry |
| Henry |
| Henry |
| Henry |
| Suzi  |
| Suzi  |
+-------+
```

To determine the number of different drivers, use `COUNT(DISTINCT)`:

```
mysql> SELECT COUNT(DISTINCT name) FROM driver_log;
+----------------------+
| COUNT(DISTINCT name) |
+----------------------+
|                    3 |
+----------------------+
```

`COUNT(DISTINCT)` ignores `NULL` values. To count `NULL` as one of the values in the set if it's present, use one of the following expressions:

```
COUNT(DISTINCT val) + IF(COUNT(IF(val IS NULL,1,NULL))=0,0,1)
COUNT(DISTINCT val) + IF(SUM(ISNULL(val))=0,0,1)
COUNT(DISTINCT val) + (SUM(ISNULL(val))<>0)
```

`DISTINCT` queries often are useful in conjunction with aggregate functions to more fully characterize your data. Suppose that a `customer` table contains a `state` column indicating customer location. Applying `COUNT(*)` to the `customer` table indicates how many customers you have, using `DISTINCT` on the `state` column tells you the number of states in which you have customers, and `COUNT(DISTINCT)` on the `state` column tells you how many states your customer base represents.

When used with multiple columns, `DISTINCT` shows the different combinations of values in the columns and `COUNT(DISTINCT)` counts the number of combinations. The

following statements show the different sender/recipient pairs in the `mail` table and the number of such pairs:

```
mysql> SELECT DISTINCT srcuser, dstuser FROM mail
    -> ORDER BY srcuser, dstuser;
+---------+---------+
| srcuser | dstuser |
+---------+---------+
| barb    | barb    |
| barb    | tricia  |
| gene    | barb    |
| gene    | gene    |
| gene    | tricia  |
| phil    | barb    |
| phil    | phil    |
| phil    | tricia  |
| tricia  | gene    |
| tricia  | phil    |
+---------+---------+
mysql> SELECT COUNT(DISTINCT srcuser, dstuser) FROM mail;
+----------------------------------+
| COUNT(DISTINCT srcuser, dstuser) |
+----------------------------------+
|                               10 |
+----------------------------------+
```

See Also

Recipe 8.2 shows how to use a view to "encapsulate" the summary expressions. Recipe 8.6 further discusses the difference between `COUNT(*)` and `COUNT(`*`expr`*`)`. The `SUM()` and `AVG()` functions are especially useful in statistical applications. They're explored further in Chapter 15, along with `STD()`, a related function that calculates standard deviations.

8.2. Creating a View to Simplify Using a Summary

Problem

You want to make it easier to perform a summary.

Solution

Create a view that does it for you.

Discussion

If you often need a given summary, a technique that enables you to avoid typing the summarizing expressions repeatedly is to use a view (see Recipe 3.7). For example, the

following view implements the weekend versus weekday trip summary discussed in Recipe 8.1:

```
mysql> CREATE VIEW trip_summary_view AS
    -> SELECT
    -> COUNT(IF(DAYOFWEEK(trav_date) IN (1,7),1,NULL)) AS weekend_trips,
    -> COUNT(IF(DAYOFWEEK(trav_date) IN (1,7),NULL,1)) AS weekday_trips
    -> FROM driver_log;
```

Selecting from this view is much easier than selecting directly from the underlying table:

```
mysql> SELECT * FROM trip_summary_view;
+---------------+---------------+
| weekend_trips | weekday_trips |
+---------------+---------------+
|             4 |             6 |
+---------------+---------------+
```

8.3. Finding Values Associated with Minimum and Maximum Values

Problem

You want to know the values for other columns in the row that contains a minimum or maximum value.

Solution

Use two statements and a user-defined variable. Or a subquery. Or a join.

Discussion

`MIN()` and `MAX()` find an endpoint of a range of values, but you may also be interested in other values from the row in which the value occurs. For example, you can find the largest state population like this:

```
mysql> SELECT MAX(pop) FROM states;
+----------+
| MAX(pop) |
+----------+
| 35893799 |
+----------+
```

But that doesn't show you which state has this population. The obvious attempt at getting that information looks like this:

```
mysql> SELECT MAX(pop), name FROM states WHERE pop = MAX(pop);
ERROR 1111 (HY000): Invalid use of group function
```

Probably everyone tries something like that sooner or later, but it doesn't work. Aggregate functions such as `MIN()` and `MAX()` cannot be used in `WHERE` clauses, which require expressions that apply to individual rows. The intent of the statement is to determine which row has the maximum population value and display the associated state name. The problem is that although you and I know perfectly well what we mean by writing such a thing, it makes no sense at all in SQL. The statement fails because SQL uses the `WHERE` clause to determine which rows to select, but the value of an aggregate function is known only *after* selecting the rows from which the function's value is determined! So, in a sense, the statement is self-contradictory. To solve this problem, save the maximum population value in a user-defined variable, then compare rows to the variable value:

```
mysql> SET @max = (SELECT MAX(pop) FROM states);
mysql> SELECT pop AS 'highest population', name FROM states WHERE pop = @max;
+--------------------+------------+
| highest population | name       |
+--------------------+------------+
|           37253956 | California |
+--------------------+------------+
```

Alternatively, for a single-statement solution, use a subquery in the `WHERE` clause that returns the maximum population value:

```
SELECT pop AS 'highest population', name FROM states
WHERE pop = (SELECT MAX(pop) FROM states);
```

This technique also works even if the minimum or maximum value itself isn't actually contained in the row, but is only derived from it. To determine the length of the shortest verse in the King James Version, do this:

```
mysql> SELECT MIN(CHAR_LENGTH(vtext)) FROM kjv;
+-------------------------+
| MIN(CHAR_LENGTH(vtext)) |
+-------------------------+
|                      11 |
+-------------------------+
```

If you want to know "Which verse is that?" do this instead:

```
mysql> SELECT bname, cnum, vnum, vtext FROM kjv
    -> WHERE CHAR_LENGTH(vtext) = (SELECT MIN(CHAR_LENGTH(vtext)) FROM kjv);
+-------+------+------+-------------+
| bname | cnum | vnum | vtext       |
+-------+------+------+-------------+
| John  |   11 |   35 | Jesus wept. |
+-------+------+------+-------------+
```

Yet another way to select other columns from rows containing a minimum or maximum value is to use a join. Select the value into another table, then join it to the original table to select the row that matches the value. To find the row for the state with the highest population, use a join like this:

```
mysql> CREATE TEMPORARY TABLE tmp SELECT MAX(pop) as maxpop FROM states;
mysql> SELECT states.* FROM states INNER JOIN tmp
    -> ON states.pop = tmp.maxpop;
+------------+--------+------------+----------+
| name       | abbrev | statehood  | pop      |
+------------+--------+------------+----------+
| California | CA     | 1850-09-09 | 37253956 |
+------------+--------+------------+----------+
```

See Also

Recipe 14.7 extends the discussion here to the problem of finding rows that contain minimum or maximum values for multiple groups in a dataset.

8.4. Controlling String Case Sensitivity for MIN() and MAX()

Problem

`MIN()` and `MAX()` select strings in case-sensitive fashion when you don't want them to, or vice versa.

Solution

Alter the comparison characteristics of the strings.

Discussion

Recipe 5.1 discusses how string-comparison properties depend on whether the strings are binary or nonbinary:

- Binary strings are sequences of bytes. They are compared byte by byte using numeric byte values. Character set and lettercase have no meaning for comparisons.
- Nonbinary strings are sequences of characters. They have a character set and collation and are compared character by character using the order defined by the collation.

These properties also apply to string columns used as the argument to the `MIN()` or `MAX()` function because they are based on comparison. To alter how these functions work with a string column, alter the column's comparison properties. Recipe 5.7 discusses how to control these properties, and Recipe 7.4 shows how they apply to string sorts. The same principles apply to finding minimum and maximum string values, so I'll just summarize here; read Recipe 7.4 for additional details.

- To compare case-insensitive strings in case-sensitive fashion, order the values using a case-sensitive collation:

```
SELECT
MIN(str_col COLLATE latin1_general_cs) AS min,
MAX(str_col COLLATE latin1_general_cs) AS max
FROM tbl;
```

- To compare case-sensitive strings in case-insensitive fashion, order the values using a case-insensitive collation:

```
SELECT
MIN(str_col COLLATE latin1_swedish_ci) AS min,
MAX(str_col COLLATE latin1_swedish_ci) AS max
FROM tbl;
```

  Another possibility is to compare values that have all been converted to the same lettercase, which makes lettercase irrelevant. However, that also changes the retrieved values:

```
SELECT
MIN(UPPER(str_col)) AS min,
MAX(UPPER(str_col)) AS max
FROM tbl;
```

- Binary strings compare using numeric byte values, so there is no concept of lettercase involved. However, because letters in different cases have different byte values, comparisons of binary strings effectively are case sensitive. (That is, `a` and `A` are unequal.) To compare binary strings using a case-insensitive ordering, convert them to nonbinary strings and apply an appropriate collation:

```
SELECT
MIN(CONVERT(str_col USING latin1) COLLATE latin1_swedish_ci) AS min,
MAX(CONVERT(str_col USING latin1) COLLATE latin1_swedish_ci) AS max
FROM tbl;
```

  If the default collation is case insensitive (as is true for `latin1`), you can omit the `COLLATE` clause.

8.5. Dividing a Summary into Subgroups

Problem

You want a summary for each subgroup of a set of rows, not an overall summary value.

Solution

Use a `GROUP BY` clause to arrange rows into groups.

Discussion

The summary statements shown so far calculate summary values over all rows in the result set. For example, the following statement determines the number of records in the `mail` table, and thus the total number of mail messages sent:

```
mysql> SELECT COUNT(*) FROM mail;
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
```

To arrange a set of rows into subgroups and summarize each group, use aggregate functions in conjunction with a `GROUP BY` clause. To determine the number of messages per sender, group the rows by sender name, count how many times each name occurs, and display the names with the counts:

```
mysql> SELECT srcuser, COUNT(*) FROM mail GROUP BY srcuser;
+---------+----------+
| srcuser | COUNT(*) |
+---------+----------+
| barb    |        3 |
| gene    |        6 |
| phil    |        5 |
| tricia  |        2 |
+---------+----------+
```

That query summarizes the same column that is used for grouping (`srcuser`), but that's not always necessary. Suppose that you want a quick characterization of the `mail` table, showing for each sender listed in it the total amount of traffic sent (in bytes) and the average number of bytes per message. In this case, you still use the `srcuser` column to group the rows, but summarize the `size` values:

```
mysql> SELECT srcuser,
    -> SUM(size) AS 'total bytes',
    -> AVG(size) AS 'bytes per message'
    -> FROM mail GROUP BY srcuser;
+---------+-------------+-------------------+
| srcuser | total bytes | bytes per message |
+---------+-------------+-------------------+
| barb    |      156696 |        52232.0000 |
| gene    |     1033108 |       172184.6667 |
| phil    |       18974 |         3794.8000 |
| tricia  |     2589407 |      1294703.5000 |
+---------+-------------+-------------------+
```

Use as many grouping columns as necessary to achieve a grouping as fine-grained as you require. The earlier query that shows the number of messages per sender is a coarse summary. To be more specific and find out how many messages each sender sent from

each host, use two grouping columns. This produces a result with nested groups (groups within groups):

```
mysql> SELECT srcuser, srchost, COUNT(srcuser) FROM mail
    -> GROUP BY srcuser, srchost;
+---------+---------+----------------+
| srcuser | srchost | COUNT(srcuser) |
+---------+---------+----------------+
| barb    | saturn  |              2 |
| barb    | venus   |              1 |
| gene    | mars    |              2 |
| gene    | saturn  |              2 |
| gene    | venus   |              2 |
| phil    | mars    |              3 |
| phil    | venus   |              2 |
| tricia  | mars    |              1 |
| tricia  | saturn  |              1 |
+---------+---------+----------------+
```

The preceding examples in this section used `COUNT()`, `SUM()`, and `AVG()` for per-group summaries. You can use `MIN()` or `MAX()`, too. With a `GROUP BY` clause, they return the smallest or largest value per group. The following query groups `mail` table rows by message sender, displaying for each the size of the largest message sent and the date of the most recent message:

```
mysql> SELECT srcuser, MAX(size), MAX(t) FROM mail GROUP BY srcuser;
+---------+-----------+---------------------+
| srcuser | MAX(size) | MAX(t)              |
+---------+-----------+---------------------+
| barb    |     98151 | 2014-05-14 14:42:21 |
| gene    |    998532 | 2014-05-19 22:21:51 |
| phil    |     10294 | 2014-05-19 12:49:23 |
| tricia  |   2394482 | 2014-05-14 17:03:01 |
+---------+-----------+---------------------+
```

You can group by multiple columns and display a maximum for each combination of values in those columns. This query finds the size of the largest message sent between each pair of sender and recipient values listed in the `mail` table:

```
mysql> SELECT srcuser, dstuser, MAX(size) FROM mail GROUP BY srcuser, dstuser;
+---------+---------+-----------+
| srcuser | dstuser | MAX(size) |
+---------+---------+-----------+
| barb    | barb    |     98151 |
| barb    | tricia  |     58274 |
| gene    | barb    |      2291 |
| gene    | gene    |     23992 |
| gene    | tricia  |    998532 |
| phil    | barb    |     10294 |
| phil    | phil    |      1048 |
| phil    | tricia  |      5781 |
| tricia  | gene    |    194925 |
```

```
| tricia  | phil    |   2394482 |
+---------+---------+-----------+
```

When using aggregate functions to produce per-group summary values, watch out for the following trap, which involves selecting nonsummary table columns not related to the grouping columns. Suppose that you want to know the longest trip per driver in the `driver_log` table:

```
mysql> SELECT name, MAX(miles) AS 'longest trip'
    -> FROM driver_log GROUP BY name;
+-------+--------------+
| name  | longest trip |
+-------+--------------+
| Ben   |          152 |
| Henry |          300 |
| Suzi  |          502 |
+-------+--------------+
```

But what if you also want to show the date on which each driver's longest trip occurred? Can you just add `trav_date` to the output column list? Sorry, that doesn't work:

```
mysql> SELECT name, trav_date, MAX(miles) AS 'longest trip'
    -> FROM driver_log GROUP BY name;
+-------+------------+--------------+
| name  | trav_date  | longest trip |
+-------+------------+--------------+
| Ben   | 2014-07-30 |          152 |
| Henry | 2014-07-29 |          300 |
| Suzi  | 2014-07-29 |          502 |
+-------+------------+--------------+
```

The query does produce a result, but if you compare it to the full table (shown here), you'll see that although the dates for Ben and Henry are correct, the date for Suzi is not:

```
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      1 | Ben   | 2014-07-30 |   152 |    ← Ben's longest trip
|      2 | Suzi  | 2014-07-29 |   391 |
|      3 | Henry | 2014-07-29 |   300 |    ← Henry's longest trip
|      4 | Henry | 2014-07-27 |    96 |
|      5 | Ben   | 2014-07-29 |   131 |
|      6 | Henry | 2014-07-26 |   115 |
|      7 | Suzi  | 2014-08-02 |   502 |    ← Suzi's longest trip
|      8 | Henry | 2014-08-01 |   197 |
|      9 | Ben   | 2014-08-02 |    79 |
|     10 | Henry | 2014-07-30 |   203 |
+--------+-------+------------+-------+
```

So what's going on? Why does the summary statement produce incorrect results? This happens because when you include a `GROUP BY` clause in a query, the only values that you can meaningfully select are the grouping columns or summary values calculated

from the groups. If you display additional table columns, they're not tied to the grouped columns and the values displayed for them are indeterminate. (For the statement just shown, it appears that MySQL may simply be picking the first date for each driver, regardless of whether it matches the driver's maximum mileage value.)

To make queries that pick indeterminate values illegal so that you won't inadvertantly suppose that the `trav_date` values are correct, set the `ONLY_FULL_GROUP_BY` SQL mode:

```
mysql> SET sql_mode = 'ONLY_FULL_GROUP_BY';
mysql> SELECT name, trav_date, MAX(miles) AS 'longest trip'
    -> FROM driver_log GROUP BY name;
ERROR 1055 (42000): 'cookbook.driver_log.trav_date' isn't in GROUP BY
```

The general solution to the problem of displaying contents of rows associated with minimum or maximum group values involves a join. The technique is described in Recipe 14.7. For the problem at hand, produce the required results as follows:

```
mysql> CREATE TEMPORARY TABLE t
    -> SELECT name, MAX(miles) AS miles FROM driver_log GROUP BY name;
mysql> SELECT d.name, d.trav_date, d.miles AS 'longest trip'
    -> FROM driver_log AS d INNER JOIN t USING (name, miles) ORDER BY name;
+-------+------------+-------------+
| name  | trav_date  | longest trip |
+-------+------------+-------------+
| Ben   | 2014-07-30 |          152 |
| Henry | 2014-07-29 |          300 |
| Suzi  | 2014-08-02 |          502 |
+-------+------------+-------------+
```

8.6. Summaries and NULL Values

Problem

You're summarizing a set of values that may include `NULL` values and you need to know how to interpret the results.

Solution

Understand how aggregate functions handle `NULL` values.

Discussion

Most aggregate functions ignore `NULL` values. `COUNT()` is different: `COUNT(`*`expr`*`)` ignores `NULL` instances of *`expr`*, but `COUNT(*)` counts rows, regardless of content.

Suppose that an `expt` table contains experimental results for subjects who are to be given four tests each and that lists the test score as `NULL` for tests not yet administered:

```
mysql> SELECT subject, test, score FROM expt ORDER BY subject, test;
+---------+------+-------+
| subject | test | score |
+---------+------+-------+
| Jane    | A    |    47 |
| Jane    | B    |    50 |
| Jane    | C    |  NULL |
| Jane    | D    |  NULL |
| Marvin  | A    |    52 |
| Marvin  | B    |    45 |
| Marvin  | C    |    53 |
| Marvin  | D    |  NULL |
+---------+------+-------+
```

By using a GROUP BY clause to arrange the rows by subject name, the number of tests taken by each subject, as well as the total, average, lowest, and highest scores, can be calculated like this:

```
mysql> SELECT subject,
    -> COUNT(score) AS n,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> MIN(score) AS lowest,
    -> MAX(score) AS highest
    -> FROM expt GROUP BY subject;
+---------+---+-------+---------+--------+---------+
| subject | n | total | average | lowest | highest |
+---------+---+-------+---------+--------+---------+
| Jane    | 2 |    97 | 48.5000 |     47 |      50 |
| Marvin  | 3 |   150 | 50.0000 |     45 |      53 |
+---------+---+-------+---------+--------+---------+
```

You can see from the results in the column labeled n (number of tests) that the query counts only five values, even though the table contains eight. Why? Because the values in that column correspond to the number of non-NULL test scores for each subject. The other summary columns display results that are calculated only from the non-NULL scores as well.

It makes a lot of sense for aggregate functions to ignore NULL values. If they followed the usual SQL arithmetic rules, adding NULL to any other value would produce a NULL result. That would make aggregate functions really difficult to use: to avoid getting a NULL result, you'd have to filter out NULL values every time you performed a summary. By ignoring NULL values, aggregate functions become a lot more convenient.

However, be aware that even though aggregate functions may ignore NULL values, some of them can still produce NULL as a result. This happens if there's nothing to summarize, which occurs if the set of values is empty or contains only NULL values. The following query is the same as the previous one, with one small difference. It selects only NULL test scores to illustrate what happens when there's nothing for the aggregate functions to operate on:

```
mysql> SELECT subject,
    -> COUNT(score) AS n,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> MIN(score) AS lowest,
    -> MAX(score) AS highest
    -> FROM expt WHERE score IS NULL GROUP BY subject;
+---------+---+-------+---------+--------+---------+
| subject | n | total | average | lowest | highest |
+---------+---+-------+---------+--------+---------+
| Jane    | 0 |  NULL |    NULL |   NULL |    NULL |
| Marvin  | 0 |  NULL |    NULL |   NULL |    NULL |
+---------+---+-------+---------+--------+---------+
```

For `COUNT()`, the number of scores per subject is zero and is reported that way. On the other hand, `SUM()`, `AVG()`, `MIN()`, and `MAX()` return `NULL` when there are no values to summarize. If you don't want an aggregate value of `NULL` to display as `NULL`, use `IFNULL()` to map it appropriately:

```
mysql> SELECT subject,
    -> COUNT(score) AS n,
    -> IFNULL(SUM(score),0) AS total,
    -> IFNULL(AVG(score),0) AS average,
    -> IFNULL(MIN(score),'Unknown') AS lowest,
    -> IFNULL(MAX(score),'Unknown') AS highest
    -> FROM expt WHERE score IS NULL GROUP BY subject;
+---------+---+-------+---------+---------+---------+
| subject | n | total | average | lowest  | highest |
+---------+---+-------+---------+---------+---------+
| Jane    | 0 |     0 |  0.0000 | Unknown | Unknown |
| Marvin  | 0 |     0 |  0.0000 | Unknown | Unknown |
+---------+---+-------+---------+---------+---------+
```

`COUNT()` is somewhat different with regard to `NULL` values than the other aggregate functions. Like other aggregate functions, `COUNT(`*`expr`*`)` counts only non-`NULL` values, but `COUNT(*)` counts rows, no matter what they contain. You can see the difference between the forms of `COUNT()` like this:

```
mysql> SELECT COUNT(*), COUNT(score) FROM expt;
+----------+--------------+
| COUNT(*) | COUNT(score) |
+----------+--------------+
|        8 |            5 |
+----------+--------------+
```

This tells us that there are eight rows in the `expt` table but that only five of them have the `score` value filled in. The different forms of `COUNT()` can be very useful for counting missing values. Just take the difference:

```
mysql> SELECT COUNT(*) - COUNT(score) AS missing FROM expt;
+---------+
| missing |
```

```
+---------+
|       3 |
+---------+
```

Missing and nonmissing counts can be determined for subgroups as well. The following query does so for each subject, providing an easy way to assess the extent to which the experiment has been completed:

```
mysql> SELECT subject,
    -> COUNT(*) AS total,
    -> COUNT(score) AS 'nonmissing',
    -> COUNT(*) - COUNT(score) AS missing
    -> FROM expt GROUP BY subject;
+---------+-------+------------+---------+
| subject | total | nonmissing | missing |
+---------+-------+------------+---------+
| Jane    |     4 |          2 |       2 |
| Marvin  |     4 |          3 |       1 |
+---------+-------+------------+---------+
```

8.7. Selecting Only Groups with Certain Characteristics

Problem

You want to calculate group summaries but display results only for groups that match certain criteria.

Solution

Use a `HAVING` clause.

Discussion

You're familiar with the use of `WHERE` to specify conditions that rows must satisfy to be selected by a statement. It's natural, therefore, to use `WHERE` to write conditions that involve summary values. The only trouble is that it doesn't work. To identify drivers in the `driver_log` table who drove more than three days, you might write the statement like this:

```
mysql> SELECT COUNT(*), name FROM driver_log
    -> WHERE COUNT(*) > 3
    -> GROUP BY name;
ERROR 1111 (HY000): Invalid use of group function
```

The problem is that `WHERE` specifies the initial constraints that determine which rows to select, but the value of `COUNT()` can be determined only after the rows have been selected. The solution is to put the `COUNT()` expression in a `HAVING` clause instead. `HAVING` is analogous to `WHERE`, but it applies to group characteristics rather than to single rows.

That is, HAVING operates on the already-selected-and-grouped set of rows, applying additional constraints based on aggregate function results that aren't known during the initial selection process. The preceding query therefore should be written like this:

```
mysql> SELECT COUNT(*), name FROM driver_log
    -> GROUP BY name
    -> HAVING COUNT(*) > 3;
+----------+-------+
| COUNT(*) | name  |
+----------+-------+
|        5 | Henry |
+----------+-------+
```

When you use HAVING, you can still include a WHERE clause, but only to select rows to be summarized, not to test already calculated summary values.

HAVING can refer to aliases, so the previous query can be rewritten like this:

```
mysql> SELECT COUNT(*) AS count, name FROM driver_log
    -> GROUP BY name
    -> HAVING count > 3;
+-------+-------+
| count | name  |
+-------+-------+
|     5 | Henry |
+-------+-------+
```

8.8. Using Counts to Determine Whether Values Are Unique

Problem

You want to know whether values in a table are unique.

Solution

Use HAVING in conjunction with COUNT().

Discussion

DISTINCT eliminates duplicates but doesn't show which values actually were duplicated in the original data. You can use HAVING to find unique values in situations to which DISTINCT does not apply. HAVING can tell you which values were unique or nonunique.

The following statements show the days on which only one driver was active, and the days on which more than one driver was active. They're based on using HAVING and COUNT() to determine which trav_date values are unique or nonunique:

```
mysql> SELECT trav_date, COUNT(trav_date) FROM driver_log
    -> GROUP BY trav_date HAVING COUNT(trav_date) = 1;
+------------+------------------+
| trav_date  | COUNT(trav_date) |
+------------+------------------+
| 2014-07-26 |                1 |
| 2014-07-27 |                1 |
| 2014-08-01 |                1 |
+------------+------------------+
mysql> SELECT trav_date, COUNT(trav_date) FROM driver_log
    -> GROUP BY trav_date HAVING COUNT(trav_date) > 1;
+------------+------------------+
| trav_date  | COUNT(trav_date) |
+------------+------------------+
| 2014-07-29 |                3 |
| 2014-07-30 |                2 |
| 2014-08-02 |                2 |
+------------+------------------+
```

This technique works for combinations of values, too. For example, to find message sender/recipient pairs between whom only one message was sent, look for combinations that occur only once in the `mail` table:

```
mysql> SELECT srcuser, dstuser FROM mail
    -> GROUP BY srcuser, dstuser HAVING COUNT(*) = 1;
+---------+---------+
| srcuser | dstuser |
+---------+---------+
| barb    | barb    |
| gene    | tricia  |
| phil    | barb    |
| tricia  | gene    |
| tricia  | phil    |
+---------+---------+
```

Note that this query doesn't print the count. The previous examples did so, to show that the counts were being used properly, but you can refer to an aggregate value in a `HAVING` clause without including it in the output column list.

8.9. Grouping by Expression Results

Problem

You want to group rows into subgroups based on values calculated from an expression.

Solution

In the `GROUP BY` clause, use an expression that categorizes values.

Discussion

GROUP BY, like ORDER BY, can refer to expressions. This means you can use calculations as the basis for grouping. As with ORDER BY, you can write the grouping expression directly in the GROUP BY clause, or use an alias for the expression (if it appears in the output column list), and refer to the alias in the GROUP BY.

To find days of the year on which more than one state joined the Union, group by statehood month and day, and then use HAVING and COUNT() to find the nonunique combinations:

```
mysql> SELECT
    -> MONTHNAME(statehood) AS month,
    -> DAYOFMONTH(statehood) AS day,
    -> COUNT(*) AS count
    -> FROM states GROUP BY month, day HAVING count > 1;
+----------+------+-------+
| month    | day  | count |
+----------+------+-------+
| February |   14 |     2 |
| June     |    1 |     2 |
| March    |    1 |     2 |
| May      |   29 |     2 |
| November |    2 |     2 |
+----------+------+-------+
```

8.10. Summarizing Noncategorical Data

Problem

You want to summarize a set of values that are not naturally categorical.

Solution

Use an expression to group the values into categories.

Discussion

Recipe 8.9 shows how to group rows by expression results. One important application for this is to categorize values that are not categorical. This is useful because GROUP BY works best for columns with repetitive values. For example, you might attempt to perform a population analysis by grouping rows in the `states` table using values in the pop column. That doesn't work very well due to the high number of distinct values in the column. In fact, they're *all* distinct:

```
mysql> SELECT COUNT(pop), COUNT(DISTINCT pop) FROM states;
```

```
+------------+--------------------+
| COUNT(pop) | COUNT(DISTINCT pop) |
+------------+--------------------+
|         50 |                 50 |
+------------+--------------------+
```

In situations like this, in which values do not group nicely into a small number of sets, use a transformation that forces them into categories. Begin by determining the range of population values:

```
mysql> SELECT MIN(pop), MAX(pop) FROM states;
+----------+----------+
| MIN(pop) | MAX(pop) |
+----------+----------+
|   563626 | 37253956 |
+----------+----------+
```

You can see from that result that if you divide the pop values by five million, they'll group into eight categories—a reasonable number. (The category ranges will be 1 to 5,000,000, 5,000,001 to 10,000,000, and so forth.) To put each population value in the proper category, divide by five million, and use the integer result:

```
mysql> SELECT FLOOR(pop/5000000) AS `max population (millions)`,
    -> COUNT(*) AS `number of states`
    -> FROM states GROUP BY `max population (millions)`;
+--------------------------+------------------+
| max population (millions) | number of states |
+--------------------------+------------------+
|                         0 |               28 |
|                         1 |               15 |
|                         2 |                3 |
|                         3 |                2 |
|                         5 |                1 |
|                         7 |                1 |
+--------------------------+------------------+
```

Hmm. That's not quite right. The expression groups the population values into a small number of categories, but doesn't report the category values properly. Let's try multiplying the FLOOR() results by five:

```
mysql> SELECT FLOOR(pop/5000000)*5 AS `max population (millions)`,
    -> COUNT(*) AS `number of states`
    -> FROM states GROUP BY `max population (millions)`;
+--------------------------+------------------+
| max population (millions) | number of states |
+--------------------------+------------------+
|                         0 |               28 |
|                         5 |               15 |
|                        10 |                3 |
|                        15 |                2 |
|                        25 |                1 |
```

```
|                         35 |                1 |
+----------------------------+------------------+
```

That still isn't correct. The maximum state population was 35,893,799, which should go into a category for 40 million, not one for 35 million. The problem here is that the category-generating expression groups values toward the lower bound of each category. To group values toward the upper bound instead, use the following technique. For categories of size *n*, place a value *x* into the proper category using this expression:

```
FLOOR((x+(n-1))/n)
```

So the final form of our query looks like this:

```
mysql> SELECT FLOOR((pop+4999999)/5000000)*5 AS `max population (millions)`,
    -> COUNT(*) AS `number of states`
    -> FROM states GROUP BY `max population (millions)`;
+----------------------------+------------------+
| max population (millions) | number of states |
+----------------------------+------------------+
|                          5 |               28 |
|                         10 |               15 |
|                         15 |                3 |
|                         20 |                2 |
|                         30 |                1 |
|                         40 |                1 |
+----------------------------+------------------+
```

The result shows clearly that the majority of US states have a population of five million or less.

In some instances, it may be more appropriate to categorize groups on a logarithmic scale. For example, treat the state population values that way as follows:

```
mysql> SELECT FLOOR(LOG10(pop)) AS `log10(population)`,
    -> COUNT(*) AS `number of states`
    -> FROM states GROUP BY `log10(population)`;
+-------------------+------------------+
| log10(population) | number of states |
+-------------------+------------------+
|                 5 |                7 |
|                 6 |               36 |
|                 7 |                7 |
+-------------------+------------------+
```

The query shows the number of states that have populations measured in hundreds of thousands, millions, and tens of millions, respectively.

You may have noticed that aliases in the preceding queries are written using backticks (identifier quoting) rather than single quotes (string quoting). Quoted aliases in the `GROUP BY` clause must use identifier quoting or the alias is treated as a constant string expression and the grouping produces the wrong result. Identifier quoting clarifies to MySQL that the alias refers to an output column. The aliases in the output column list

could have been written using string quoting; I used backticks there to avoid mixing alias quoting styles within a given query.

How Repetitive Is a Set of Values?

To assess how much repetition is present in a set of values, use the ratio of `COUNT(DISTINCT)` and `COUNT()`. If all values are unique, both counts are the same and the ratio is 1. This is the case for the `t` values in the `mail` table and the `pop` values in the `states` table:

```
mysql> SELECT COUNT(DISTINCT t) / COUNT(t) FROM mail;
+-----------------------------+
| COUNT(DISTINCT t) / COUNT(t) |
+-----------------------------+
|                      1.0000 |
+-----------------------------+
mysql> SELECT COUNT(DISTINCT pop) / COUNT(pop) FROM states;
+---------------------------------+
| COUNT(DISTINCT pop) / COUNT(pop) |
+---------------------------------+
|                           1.0000 |
+---------------------------------+
```

For a more repetitive set of values, `COUNT(DISTINCT)` is less than `COUNT()`, and the ratio is smaller:

```
mysql> SELECT COUNT(DISTINCT name) / COUNT(name) FROM driver_log;
+-----------------------------------+
| COUNT(DISTINCT name) / COUNT(name) |
+-----------------------------------+
|                            0.3000 |
+-----------------------------------+
```

What's the practical use for this ratio? A result close to zero indicates a high degree of repetition, which means the values will group into a small number of categories naturally. A result of 1 or close to it indicates many unique values, with the consequence that `GROUP BY` won't be very efficient for grouping the values into categories. (That is, there will be a lot of categories, relative to the number of values.) This tells you that, to generate a summary, you'll probably find it necessary to impose an artificial categorization on the values, using the techniques described in this recipe.

8.11. Finding Smallest or Largest Summary Values

Problem

You want to compute per-group summary values but display only the smallest or largest of them.

Solution

Add a `LIMIT` clause to the statement. Or use a user-defined variable or subquery to pick the appropriate summary.

Discussion

`MIN()` and `MAX()` find the values at the endpoints of a set of values, but to find the endpoints of a set of summary values, those functions won't work. Their argument cannot be another aggregate function. For example, you can easily find per-driver mileage totals:

```
mysql> SELECT name, SUM(miles)
    -> FROM driver_log
    -> GROUP BY name;
+-------+------------+
| name  | SUM(miles) |
+-------+------------+
| Ben   |        362 |
| Henry |        911 |
| Suzi  |        893 |
+-------+------------+
```

To select only the row for the driver with the most miles, the following doesn't work:

```
mysql> SELECT name, SUM(miles)
    -> FROM driver_log
    -> GROUP BY name
    -> HAVING SUM(miles) = MAX(SUM(miles));
ERROR 1111 (HY000): Invalid use of group function
```

Instead, order the rows with the largest `SUM()` values first and use `LIMIT` to select the first row:

```
mysql> SELECT name, SUM(miles)
    -> FROM driver_log
    -> GROUP BY name
    -> ORDER BY SUM(miles) DESC LIMIT 1;
+-------+------------+
| name  | SUM(miles) |
+-------+------------+
| Henry |        911 |
+-------+------------+
```

However, if more than one row has the given summary value, a `LIMIT 1` query won't tell you that. For example, you might attempt to ascertain the most common initial letter for state names like this:

```
mysql> SELECT LEFT(name,1) AS letter, COUNT(*) FROM states
    -> GROUP BY letter ORDER BY COUNT(*) DESC LIMIT 1;
+--------+----------+
| letter | COUNT(*) |
```

```
+--------+----------+
| N      |        8 |
+--------+----------+
```

But eight state names also begin with `N`. To find all most-frequent values when there may be more than one, use a user-defined variable or subquery to determine the maximum count, then select those values with a count equal to the maximum:

```
mysql> SET @max = (SELECT COUNT(*) FROM states
    -> GROUP BY LEFT(name,1) ORDER BY COUNT(*) DESC LIMIT 1);
mysql> SELECT LEFT(name,1) AS letter, COUNT(*) FROM states
    -> GROUP BY letter HAVING COUNT(*) = @max;
+--------+----------+
| letter | COUNT(*) |
+--------+----------+
| M      |        8 |
| N      |        8 |
+--------+----------+
mysql> SELECT LEFT(name,1) AS letter, COUNT(*) FROM states
    -> GROUP BY letter HAVING COUNT(*) =
    ->   (SELECT COUNT(*) FROM states
    ->   GROUP BY LEFT(name,1) ORDER BY COUNT(*) DESC LIMIT 1);
+--------+----------+
| letter | COUNT(*) |
+--------+----------+
| M      |        8 |
| N      |        8 |
+--------+----------+
```

8.12. Date-Based Summaries

Problem

You want to produce a summary based on date or time values.

Solution

Use `GROUP BY` to place temporal values into categories of the appropriate duration. Often this involves using expressions that extract the significant parts of dates or times.

Discussion

To sort rows temporally, use `ORDER BY` with a temporal column. To summarize rows instead, based on groupings into time intervals, determine how to categorize rows into the proper intervals and use `GROUP BY` to group them accordingly.

For example, to determine how many drivers were on the road and how many miles were driven each day, group the rows in the `driver_log` table by date:[1]

```
mysql> SELECT trav_date,
    -> COUNT(*) AS 'number of drivers', SUM(miles) As 'miles logged'
    -> FROM driver_log GROUP BY trav_date;
+------------+------------------+--------------+
| trav_date  | number of drivers | miles logged |
+------------+------------------+--------------+
| 2014-07-26 |                1 |          115 |
| 2014-07-27 |                1 |           96 |
| 2014-07-29 |                3 |          822 |
| 2014-07-30 |                2 |          355 |
| 2014-08-01 |                1 |          197 |
| 2014-08-02 |                2 |          581 |
+------------+------------------+--------------+
```

However, this per-day summary grows lengthier as you add more rows to the table. Over time, the number of distinct dates will become so large that the summary fails to be useful, and you'd probably decide to increase the category size. For example, this query categorizes by month:

```
mysql> SELECT YEAR(trav_date) AS year, MONTH(trav_date) AS month,
    -> COUNT(*) AS 'number of drivers', SUM(miles) As 'miles logged'
    -> FROM driver_log GROUP BY year, month;
+------+-------+------------------+--------------+
| year | month | number of drivers | miles logged |
+------+-------+------------------+--------------+
| 2014 |     7 |                7 |         1388 |
| 2014 |     8 |                3 |          778 |
+------+-------+------------------+--------------+
```

Now the number of summary rows grows much more slowly over time. Eventually, you could summarize based only on year to collapse rows even more.

Uses for temporal categorizations are numerous:

- To produce daily summaries from `DATETIME` or `TIMESTAMP` columns that have the potential to contain many unique values, strip the time-of-day part to collapse all values occurring within a given day to the same value. Any of the following `GROUP BY` clauses will do this, although the last one is likely to be slowest:

```
GROUP BY DATE(col_name)
GROUP BY FROM_DAYS(TO_DAYS(col_name))
GROUP BY YEAR(col_name), MONTH(col_name), DAYOFMONTH(col_name)
GROUP BY DATE_FORMAT(col_name,'%Y-%m-%e')
```

1. The result includes an entry only for dates actually represented in the table. To generate a summary with an entry for the range of dates in the table, use a join to fill in the "missing" values. See Recipe 14.8.

- To produce monthly or quarterly sales reports, group by `MONTH(col_name)` or `QUARTER(col_name)` to place dates into the correct part of the year.
- To summarize web server activity, store your server's logs in MySQL and run statements that collapse the rows into different time categories. Recipe 20.14 discusses how to do this for Apache.

8.13. Working with Per-Group and Overall Summary Values Simultaneously

Problem

You want to produce a report that requires different levels of summary detail. Or you want to compare per-group summary values to an overall summary value.

Solution

Use two statements that retrieve different levels of summary information. Or use a subquery to retrieve one summary value and refer to it in the outer query that refers to other summary values. For applications that only display multiple summary levels (rather than perform additional calculations on them), `WITH ROLLUP` might be sufficient.

Discussion

Some reports involve multiple levels of summary information. The following report displays the total number of miles per driver from the `driver_log` table, along with each driver's miles as a percentage of the total miles in the entire table:

```
+-------+-------------+-----------------------+
| name  | miles/driver | percent of total miles |
+-------+-------------+-----------------------+
| Ben   |          362 |                16.7128 |
| Henry |          911 |                42.0591 |
| Suzi  |          893 |                41.2281 |
+-------+-------------+-----------------------+
```

The percentages represent the ratio of each driver's miles to the total miles for all drivers. To perform the percentage calculation, you need a per-group summary to get each driver's miles and also an overall summary to get the total miles. First, run a query to get the overall mileage total:

```
mysql> SELECT @total := SUM(miles) AS 'total miles' FROM driver_log;
+-------------+
| total miles |
+-------------+
```

```
|        2166 |
+-------------+
```

Then calculate the per-group values and use the overall total to compute the percentages:

```
mysql> SELECT name,
    -> SUM(miles) AS 'miles/driver',
    -> (SUM(miles)*100)/@total AS 'percent of total miles'
    -> FROM driver_log GROUP BY name;
+-------+-------------+-----------------------+
| name  | miles/driver | percent of total miles |
+-------+-------------+-----------------------+
| Ben   |          362 |                16.7128 |
| Henry |          911 |                42.0591 |
| Suzi  |          893 |                41.2281 |
+-------+-------------+-----------------------+
```

To combine the two statements into one, use a subquery that computes the total miles:

```
SELECT name,
SUM(miles) AS 'miles/driver',
(SUM(miles)*100)/(SELECT SUM(miles) FROM driver_log)
  AS 'percent of total miles'
FROM driver_log GROUP BY name;
```

A similar problem uses multiple summary levels to compare per-group summary values with the corresponding overall summary value. Suppose that you want to display drivers who had a lower average miles per day than the group average. Calculate the overall average in a subquery, and then compare each driver's average to the overall average using a `HAVING` clause:

```
mysql> SELECT name, AVG(miles) AS driver_avg FROM driver_log
    -> GROUP BY name
    -> HAVING driver_avg < (SELECT AVG(miles) FROM driver_log);
+-------+------------+
| name  | driver_avg |
+-------+------------+
| Ben   |   120.6667 |
| Henry |   182.2000 |
+-------+------------+
```

To display different summary-level values (and not perform calculations involving one summary level against another), add `WITH ROLLUP` to the `GROUP BY` clause:

```
mysql> SELECT name, SUM(miles) AS 'miles/driver'
    -> FROM driver_log GROUP BY name WITH ROLLUP;
+-------+-------------+
| name  | miles/driver |
+-------+-------------+
| Ben   |          362 |
| Henry |          911 |
| Suzi  |          893 |
| NULL  |         2166 |
+-------+-------------+
```

```
mysql> SELECT name, AVG(miles) AS driver_avg FROM driver_log
    -> GROUP BY name WITH ROLLUP;
+-------+------------+
| name  | driver_avg |
+-------+------------+
| Ben   |   120.6667 |
| Henry |   182.2000 |
| Suzi  |   446.5000 |
| NULL  |   216.6000 |
+-------+------------+
```

In each case, the output row with `NULL` in the `name` column represents the overall sum or average calculated over all drivers.

`WITH ROLLUP` produces multiple summary levels if you group by more than one column. The following statement shows the number of mail messages sent between each pair of users:

```
mysql> SELECT srcuser, dstuser, COUNT(*)
    -> FROM mail GROUP BY srcuser, dstuser;
+---------+---------+----------+
| srcuser | dstuser | COUNT(*) |
+---------+---------+----------+
| barb    | barb    |        1 |
| barb    | tricia  |        2 |
| gene    | barb    |        2 |
| gene    | gene    |        3 |
| gene    | tricia  |        1 |
| phil    | barb    |        1 |
| phil    | phil    |        2 |
| phil    | tricia  |        2 |
| tricia  | gene    |        1 |
| tricia  | phil    |        1 |
+---------+---------+----------+
```

Adding `WITH ROLLUP` causes the output to include an intermediate count for each `srcuser` value (these are the lines with `NULL` in the `dstuser` column), plus an overall count at the end:

```
mysql> SELECT srcuser, dstuser, COUNT(*)
    -> FROM mail GROUP BY srcuser, dstuser WITH ROLLUP;
+---------+---------+----------+
| srcuser | dstuser | COUNT(*) |
+---------+---------+----------+
| barb    | barb    |        1 |
| barb    | tricia  |        2 |
| barb    | NULL    |        3 |
| gene    | barb    |        2 |
| gene    | gene    |        3 |
| gene    | tricia  |        1 |
| gene    | NULL    |        6 |
| phil    | barb    |        1 |
```

```
| phil    | phil    |        2 |
| phil    | tricia  |        2 |
| phil    | NULL    |        5 |
| tricia  | gene    |        1 |
| tricia  | phil    |        1 |
| tricia  | NULL    |        2 |
| NULL    | NULL    |       16 |
+---------+---------+----------+
```

8.14. Generating a Report That Includes a Summary and a List

Problem

You want to create a report that displays a summary, together with the list of rows associated with each summary value.

Solution

Use two statements that retrieve different levels of summary information. Or use a programming language to do some of the work so that you can use a single statement.

Discussion

Suppose that you want to produce a report that looks like this:

```
Name: Ben; days on road: 3; miles driven: 362
  date: 2014-07-29, trip length: 131
  date: 2014-07-30, trip length: 152
  date: 2014-08-02, trip length: 79
Name: Henry; days on road: 5; miles driven: 911
  date: 2014-07-26, trip length: 115
  date: 2014-07-27, trip length: 96
  date: 2014-07-29, trip length: 300
  date: 2014-07-30, trip length: 203
  date: 2014-08-01, trip length: 197
Name: Suzi; days on road: 2; miles driven: 893
  date: 2014-07-29, trip length: 391
  date: 2014-08-02, trip length: 502
```

For each driver in the `driver_log` table, the report shows the following information:

- A summary line showing the driver name, the number of days on the road, and the number of miles driven.
- A list that details dates and mileages for the individual trips from which the summary values are calculated.

This scenario is a variation on the "different levels of summary information" problem discussed in Recipe 8.13. It may not seem like it at first because one of the types of information is a list rather than a summary. But that's really just a "level zero" summary. This kind of problem appears in many other forms:

- You have a database that lists contributions to candidates in your political party. The party chair requests a printout that shows, for each candidate, the number of contributions and total amount contributed, as well as a list of contributor names and addresses.
- You want to create a handout for a company presentation that summarizes total sales per sales region with a list under each region showing the sales for each state in the region.

Such problems have multiple solutions:

- Run separate statements to get the information for each level of detail that you require. (A single query won't produce per-group summary values and a list of each group's individual rows.)
- Fetch the rows that make up the lists and perform the summary calculations yourself to eliminate the summary statement.

Let's use each approach to produce the driver report shown at the beginning of this section. The following implementation (in Python) generates the report using one query to summarize the days and miles per driver, and another to fetch the individual trip rows for each driver:

```
# select total miles per driver and construct a dictionary that
# maps each driver name to days on the road and miles driven
name_map = {}
cursor = conn.cursor()
cursor.execute('''
               SELECT name, COUNT(name), SUM(miles)
               FROM driver_log GROUP BY name
               ''')
for (name, days, miles) in cursor:
  name_map[name] = (days, miles)

# select trips for each driver and print the report, displaying the
# summary entry for each driver prior to the list of trips
cursor.execute('''
               SELECT name, trav_date, miles
               FROM driver_log ORDER BY name, trav_date
               ''')
cur_name = ""
for (name, trav_date, miles) in cursor:
  if cur_name != name:  # new driver; print driver's summary info
    print("Name: %s; days on road: %d; miles driven: %d" %
```

```
            (name, name_map[name][0], name_map[name][1]))
      cur_name = name
    print("  date: %s, trip length: %d" % (trav_date, miles))
cursor.close()
```

An alternative implementation performs summary calculations within the program, which reduces the number of queries required. If you iterate through the trip list and calculate the per-driver day counts and mileage totals yourself, a single query suffices:

```
# get list of trips for the drivers
cursor = conn.cursor()
cursor.execute('''
               SELECT name, trav_date, miles FROM driver_log
               ORDER BY name, trav_date
               ''')
# fetch rows into data structure because we
# must iterate through them multiple times
rows = cursor.fetchall()
cursor.close()

# iterate through rows once to construct a dictionary that
# maps each driver name to days on the road and miles driven
# (the dictionary entries are lists rather than tuples because
# we need mutable values that can be modified in the loop)
name_map = {}
for (name, trav_date, miles) in rows:
  if not name_map.has_key(name): # initialize entry if nonexistent
    name_map[name] = [0, 0]
  name_map[name][0] += 1     # count days
  name_map[name][1] += miles # sum miles

# iterate through rows again to print the report, displaying the
# summary entry for each driver prior to the list of trips
cur_name = ""
for (name, trav_date, miles) in rows:
  if cur_name != name:  # new driver; print driver's summary info
    print("Name: %s; days on road: %d; miles driven: %d" %
          (name, name_map[name][0], name_map[name][1]))
    cur_name = name
  print("  date: %s, trip length: %d" % (trav_date, miles))
```

Should you require more levels of summary information, this type of problem gets more difficult. For example, you might want to precede the report that shows driver summaries and trip logs with a line that shows the total miles for all drivers:

```
Total miles driven by all drivers combined: 2166

Name: Ben; days on road: 3; miles driven: 362
  date: 2014-07-29, trip length: 131
  date: 2014-07-30, trip length: 152
  date: 2014-08-02, trip length: 79
Name: Henry; days on road: 5; miles driven: 911
  date: 2014-07-26, trip length: 115
```

```
  date: 2014-07-27, trip length: 96
  date: 2014-07-29, trip length: 300
  date: 2014-07-30, trip length: 203
  date: 2014-08-01, trip length: 197
Name: Suzi; days on road: 2; miles driven: 893
  date: 2014-07-29, trip length: 391
  date: 2014-08-02, trip length: 502
```

In this case, you need either another query to produce the total mileage, or another calculation in your program that computes the overall total.

CHAPTER 9

Using Stored Routines, Triggers, and Scheduled Events

9.0. Introduction

This chapter discusses stored database objects, which come in several varieties:

Stored functions and procedures
: A stored function or procedure object encapsulates the code for performing an operation, enabling you to invoke the object easily by name rather than repeat all its code each time it's needed. A stored function performs a calculation and returns a value that can be used in expressions just like a built-in function such as `RAND()`, `NOW()`, or `LEFT()`. A stored procedure performs operations for which no return value is needed. Procedures are invoked with the `CALL` statement, not used in expressions. A procedure might update rows in a table or produce a result set that is sent to the client program.

Triggers
: A trigger is an object that activates when a table is modified by an `INSERT`, `UPDATE`, or `DELETE` statement. For example, you can check values before they are inserted into a table, or specify that any row deleted from a table should be logged to another table that serves as a journal of data changes. Triggers automate these actions so that you need not remember to do them yourself each time you modify a table.

Scheduled events
: An event is an object that executes SQL statements at a scheduled time or times. Think of a scheduled event as something like a Unix *cron* job that runs within MySQL. For example, events can help you perform administrative tasks such as deleting old table rows periodically or creating nightly summaries.

In this book, the term "stored routine" refers collectively to stored functions and procedures, and "stored program" refers collectively to stored routines, triggers, and events.

Stored programs are database objects that are user-defined but stored on the server side for later execution. This differs from sending an SQL statement from the client to the server for immediate execution. Each object also has the property that it is defined in terms of other SQL statements to be executed when the object is invoked. The object body is a single SQL statement, but that statement can use compound-statement syntax (a `BEGIN ... END` block) that contains multiple statements. Thus, the body can range from very simple to extremely complex. The following stored procedure is a trivial routine that does nothing but display the current MySQL version, using a body that consists of a single `SELECT` statement:

```
CREATE PROCEDURE show_version()
SELECT VERSION() AS 'MySQL Version';
```

More complex operations use a `BEGIN ... END` compound statement:

```
CREATE PROCEDURE show_part_of_day()
BEGIN
  DECLARE cur_time, day_part TEXT;
  SET cur_time = CURTIME();
  IF cur_time < '12:00:00' THEN
    SET day_part = 'morning';
  ELSEIF cur_time = '12:00:00' THEN
    SET day_part = 'noon';
  ELSE
    SET day_part = 'afternoon or night';
  END IF;
  SELECT cur_time, day_part;
END;
```

Here, the `BEGIN ... END` block contains multiple statements, but is itself considered to constitute a single statement. Compound statements enable you to declare local variables and to use conditional logic and looping constructs. These capabilities provide considerably more flexibility for algorithmic expression than when you write inline expressions in noncompound statements such as `SELECT` or `UPDATE`.

Each statement within a compound statement must be terminated by a `;` character. That requirement causes a problem if you use the *mysql* client to define an object that uses compound statements because *mysql* itself interprets `;` to determine statement boundaries. The solution is to redefine *mysql*'s statement delimiter while you define a compound-statement object. Recipe 9.1 covers how to do this; be sure to read that recipe before proceeding to those that follow it.

This chapter illustrates stored routines, triggers, and events by example, but due to space limitations does not otherwise go into much detail about their extensive syntax. For complete syntax descriptions, see the *MySQL Reference Manual*.

Scripts for the examples shown in this chapter are located in the *routines*, *triggers*, and *events* directories of the `recipes` distribution. Scripts to create example tables are located in the *tables* directory.

In addition to the stored programs shown in this chapter, others can be found elsewhere in this book. See, for example, Recipes 5.6, 6.3, 14.8, and 23.2.

Stored programs used here are created and invoked under the assumption that `cookbook` is the default database. To invoke a program from another database, qualify its name with the database name:

```
CALL cookbook.show_version();
```

Alternatively, create a database specifically for your stored programs, create them in that database, and always invoke them qualified with that name. Remember to grant users who will use them the `EXECUTE` privilege for that database.

Privileges for Stored Programs

When you create a stored routine (function or procedure), the following privilege requirements must be satisfied or you will have problems:

- To create or execute the routine, you must have the `CREATE ROUTINE` or `EXECUTE` privilege, respectively.
- If binary logging is enabled for your MySQL server, as is common practice, there are additional requirements for creating stored functions (but not stored procedures). These requirements are necessary to ensure that if you use the binary log for replication or for restoring backups, function invocations cause the same effect when re-executed as they do when originally executed:
  - You must have the `SUPER` privilege, and you must declare either that the function is deterministic or does not modify data by using one of the `DETERMINISTIC`, `NO SQL`, or `READS SQL DATA` characteristics. (It's possible to create functions that are not deterministic or that modify data, but they might not be safe for replication or for use in backups.)
  - Alternatively, if you enable the `log_bin_trust_function_creators` system variable, the server waives both of the preceding requirements. You can do this at server startup, or at runtime if you have the `SUPER` privilege.

To create a trigger, you must have the `TRIGGER` privilege for the table associated with the trigger.

To create a scheduled event, you must have the `EVENT` privilege for the database in which the event is created.

For information about granting privileges, see Recipe 23.2.

9.1. Creating Compound-Statement Objects

Problem

You want to define a stored program, but its body contains instances of the `;` statement terminator. The *mysql* client program uses the same terminator by default, so *mysql* misinterprets the definition and produces an error.

Solution

Redefine the *mysql* statement terminator with the `delimiter` command.

Discussion

Each stored program is an object with a body that must be a single SQL statement. However, these objects often perform complex operations that require several statements. To handle this, write the statements within a `BEGIN ... END` block that forms a compound statement. That is, the block is itself a single statement but can contain multiple statements, each terminated by a `;` character. The `BEGIN ... END` block can contain statements such as `SELECT` or `INSERT`, but compound statements also permit conditional statements such as `IF` or `CASE`, looping constructs such as `WHILE` or `REPEAT`, or other `BEGIN ... END` blocks.

Compound-statement syntax provides flexibility, but if you define compound-statement objects within the *mysql* client, you quickly encounter a problem: each statement within a compound statement must be terminated by a `;` character, but *mysql* itself interprets `;` to figure out where statements end so that it can send them one at a time to the server to be executed. Consequently, *mysql* stops reading the compound statement when it sees the first `;` character, which is too early. To handle this, tell *mysql* to recognize a different statement delimiter so that it ignores `;` characters within the object body. Terminate the object itself with the new delimiter, which *mysql* recognizes and then sends the entire object definition to the server. You can restore the *mysql* delimiter to its original value after defining the compound-statement object.

The following example uses a stored function to illustrate how to change the delimiter, but the principles apply to defining any type of stored program.

Suppose that you want to create a stored function that calculates and returns the average size in bytes of mail messages listed in the `mail` table. The function can be defined like this, where the body consists of a single SQL statement:

```
CREATE FUNCTION avg_mail_size()
RETURNS FLOAT READS SQL DATA
RETURN (SELECT AVG(size) FROM mail);
```

The `RETURNS FLOAT` clause indicates the type of the function's return value, and `READS SQL DATA` indicates that the function reads but does not modify data. The function body follows those clauses: a single `RETURN` statement that executes a subquery and returns the resulting value to the caller. (Every stored function must have at least one `RETURN` statement.)

In *mysql*, you can enter that statement as shown and there is no problem. The definition requires just the single terminator at the end and none internally, so no ambiguity arises. But suppose instead that you want the function to take an argument naming a user that it interprets as follows:

- If the argument is `NULL`, the function returns the average size for all messages (as before).
- If the argument is non-`NULL`, the function returns the average size for messages sent by that user.

To accomplish this, the function has a more complex body that uses a `BEGIN ... END` block:

```
CREATE FUNCTION avg_mail_size(user VARCHAR(8))
RETURNS FLOAT READS SQL DATA
BEGIN
  DECLARE avg FLOAT;
  IF user IS NULL
  THEN # average message size over all users
    SET avg = (SELECT AVG(size) FROM mail);
  ELSE # average message size for given user
    SET avg = (SELECT AVG(size) FROM mail WHERE srcuser = user);
  END IF;
  RETURN avg;
END;
```

If you try to define the function within *mysql* by entering that definition as just shown, *mysql* improperly interprets the first semicolon in the function body as ending the definition. Instead, use the `delimiter` command to change the *mysql* delimiter, then restore the delimiter to its default value:

```
mysql> delimiter $$
mysql> CREATE FUNCTION avg_mail_size(user VARCHAR(8))
    -> RETURNS FLOAT READS SQL DATA
    -> BEGIN
```

```
    ->   DECLARE avg FLOAT;
    ->   IF user IS NULL
    ->   THEN # average message size over all users
    ->     SET avg = (SELECT AVG(size) FROM mail);
    ->   ELSE # average message size for given user
    ->     SET avg = (SELECT AVG(size) FROM mail WHERE srcuser = user);
    ->   END IF;
    ->   RETURN avg;
    -> END;
    -> $$
Query OK, 0 rows affected (0.02 sec)
mysql> delimiter ;
```

After defining the stored function, invoke it the same way as a built-in function:

```
mysql> SELECT avg_mail_size(NULL), avg_mail_size('barb');
+---------------------+-----------------------+
| avg_mail_size(NULL) | avg_mail_size('barb') |
+---------------------+-----------------------+
|         237386.5625 |                 52232 |
+---------------------+-----------------------+
```

9.2. Using Stored Functions to Encapsulate Calculations

Problem

A particular calculation to produce a value must be performed frequently by different applications, but you don't want to write the expression for it each time it's needed. Or a calculation is difficult to perform inline within an expression because it requires conditional or looping logic.

Solution

Use a stored function to hide the ugly details and make the calculation easy to perform.

Discussion

Stored functions enable you to simplify your applications. Write the code that produces a calculation result once in a function definition, then simply invoke the function whenever you need to perform the calculation. Stored functions also enable you to use more complex algorithmic constructs than are available when you write a calculation inline within an expression. This section illustrates how stored functions can be useful in these ways. (Granted, the example is not *that* complex, but the principles used here apply to writing much more elaborate functions.)

Different states in the US charge different rates for sales tax. If you sell goods to people from different states, you must charge tax using the rate appropriate for customer state

of residence. To handle tax computations, use a table that lists the sales tax rate for each state, and a stored function that looks up the tax rate given a state.

To set up the `sales_tax_rate` table, use the *sales_tax_rate.sql* script in the *tables* directory of the `recipes` distribution. The table has two columns: `state` (a two-letter abbreviation), and `tax_rate` (a `DECIMAL` value rather than a `FLOAT`, to preserve accuracy).

Define the rate-lookup function, `sales_tax_rate()`, as follows:

```
CREATE FUNCTION sales_tax_rate(state_code CHAR(2))
RETURNS DECIMAL(3,2) READS SQL DATA
BEGIN
  DECLARE rate DECIMAL(3,2);
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET rate = 0;
  SELECT tax_rate INTO rate FROM sales_tax_rate WHERE state = state_code;
  RETURN rate;
END;
```

Suppose that the tax rates for Vermont and New York are 1 and 9 percent, respectively. Try the function to check whether the tax rate is returned correctly:

```
mysql> SELECT sales_tax_rate('VT'), sales_tax_rate('NY');
+----------------------+----------------------+
| sales_tax_rate('VT') | sales_tax_rate('NY') |
+----------------------+----------------------+
|                 0.01 |                 0.09 |
+----------------------+----------------------+
```

If you take sales from a location not listed in the table, the function cannot determine the rate for it. In this case, the function assumes a tax rate of 0 percent:

```
mysql> SELECT sales_tax_rate('ZZ');
+----------------------+
| sales_tax_rate('ZZ') |
+----------------------+
|                 0.00 |
+----------------------+
```

The function handles states not listed using a `CONTINUE` handler for `NOT FOUND`, which executes if a No Data condition occurs: if there is no row for the given `state_param` value, the `SELECT` statement fails to find a sales tax rate, the `CONTINUE` handler sets the rate to 0, and continues execution with the next statement after the `SELECT`. (This handler is an example of stored routine logic not available in inline expressions. Recipe 9.10 discusses handlers further.)

To compute sales tax for a purchase, multiply the purchase price by the tax rate. For example, for Vermont and New York, tax on a $150 purchase is:

```
mysql> SELECT 150*sales_tax_rate('VT'), 150*sales_tax_rate('NY');
```

```
+-------------------------+-------------------------+
| 150*sales_tax_rate('VT') | 150*sales_tax_rate('NY') |
+-------------------------+-------------------------+
|                    1.50 |                   13.50 |
+-------------------------+-------------------------+
```

Or write another function that computes the tax for you:

```
CREATE FUNCTION sales_tax(state_code CHAR(2), sales_amount DECIMAL(10,2))
RETURNS DECIMAL(10,2) READS SQL DATA
RETURN sales_amount * sales_tax_rate(state_code);
```

And use it like this:

```
mysql> SELECT sales_tax('VT',150), sales_tax('NY',150);
+--------------------+--------------------+
| sales_tax('VT',150) | sales_tax('NY',150) |
+--------------------+--------------------+
|               1.50 |              13.50 |
+--------------------+--------------------+
```

9.3. Using Stored Procedures to "Return" Multiple Values

Problem

An operation produces two or more values, but a stored function can return only a single value.

Solution

Use a stored procedure that has `OUT` or `INOUT` parameters, and pass user-defined variables for those parameters when you invoke the procedure. A procedure does not "return" a value the way a function does, but it can assign values to those parameters so that the user-defined variables have the desired values when the procedure returns.

Discussion

Unlike stored function parameters, which are input values only, a stored procedure parameter can be any of three types:

- An `IN` parameter is for input only. This is the default if you specify no type.
- An `INOUT` parameter is used to pass a value in, and can also pass a value out.
- An `OUT` parameter is used to pass a value out.

Thus, to produce multiple values from an operation, you can use `INOUT` or `OUT` parameters. The following example illustrates this, using an `IN` parameter for input, and passing back three values via `OUT` parameters.

Recipe 9.1 shows an `avg_mail_size()` function that returns the average mail message size for a given sender. The function returns a single value. To produce additional information, such as the number of messages and total message size, a function will not work. You could write three separate functions, but that is cumbersome. Instead, use a single procedure that retrieves multiple values about a given mail sender. The following procedure, `mail_sender_stats()`, runs a query on the `mail` table to retrieve mail-sending statistics about a given username, which is the input value. The procedure determines how many messages that user sent, and the total and average sizes of the messages in bytes, which it returns through three `OUT` parameters:

```
CREATE PROCEDURE mail_sender_stats(IN user VARCHAR(8),
                                   OUT messages INT,
                                   OUT total_size INT,
                                   OUT avg_size INT)
BEGIN
  # Use IFNULL() to return 0 for SUM() and AVG() in case there are
  # no rows for the user (those functions return NULL in that case).
  SELECT COUNT(*), IFNULL(SUM(size),0), IFNULL(AVG(size),0)
  INTO messages, total_size, avg_size
  FROM mail WHERE srcuser = user;
END;
```

To use the procedure, pass a string containing the username, and three user-defined variables to receive the `OUT` values. After the procedure returns, access the variable values:

```
mysql> CALL mail_sender_stats('barb',@messages,@total_size,@avg_size);
mysql> SELECT @messages, @total_size, @avg_size;
+-----------+-------------+-----------+
| @messages | @total_size | @avg_size |
+-----------+-------------+-----------+
|         3 |      156696 |     52232 |
+-----------+-------------+-----------+
```

This routine passes back calculation results. It's also common to use `OUT` parameters for diagnostic purposes such as status or error indicators.

If you call `mail_sender_stats()` from within a stored program, you can pass variables to it using routine parameters or program local variables, not just user-defined variables.

9.4. Using Triggers to Implement Dynamic Default Column Values

Problem

A table contains a column for which the initial value is not constant, but in most cases, MySQL permits only constant default values.

Solution

Use a BEFORE INSERT trigger. This enables you to initialize a column to the value of an arbitrary expression. In other words, the trigger performs dynamic column initialization by calculating the default value.

Discussion

Other than TIMESTAMP and DATETIME columns, which can be initialized to the current date and time (see Recipe 6.7), default column values in MySQL must be constants. You cannot define a column with a DEFAULT clause that refers to a function call or other arbitrary expression, and you cannot define one column in terms of the value assigned to another column. That means each of these column definitions is illegal:

```
d        DATE DEFAULT NOW()
i        INT DEFAULT (... some subquery ...)
hash_val CHAR(32) DEFAULT MD5(blob_col)
```

You can work around this limitation by setting up a suitable trigger, which enables you to initialize a column however you want. In effect, the trigger implements a dynamic (or calculated) default column value.

The appropriate type of trigger for this is BEFORE INSERT, which enables column values to be set before they are inserted into the table. (An AFTER INSERT trigger can examine column values for a new row, but by the time the trigger activates, it's too late to change the values.)

To see how this works, recall the scenario in Recipe 9.2 that created a sales_tax_rate() lookup function to return a rate from the sales_tax_rate table given a customer state of residence. Suppose that you anticipate a need to know at some later date the tax rate from the time of sale. It's not necessarily true that at that later date you could look up the value from the sales_tax_rate table; rates change and the rate in effect then might differ. To handle this, store the rate with the purchase invoice, initializing it automatically using a trigger.

A cust_invoice table for storing sales information might look like this:

```
CREATE TABLE cust_invoice
(
  id       INT NOT NULL AUTO_INCREMENT,
  state    CHAR(2),       # customer state of residence
  amount   DECIMAL(10,2), # sale amount
  tax_rate DECIMAL(3,2),  # sales tax rate at time of purchase
  ... other columns ...
  PRIMARY KEY (id)
);
```

To initialize the sales tax column for inserts into the cust_invoice table, use a BEFORE INSERT trigger that looks up the rate and stores it in the table:

```
CREATE TRIGGER bi_cust_invoice BEFORE INSERT ON cust_invoice
FOR EACH ROW SET NEW.tax_rate = sales_tax_rate(NEW.state);
```

Within the trigger, `NEW.col_name` refers to the new value to be inserted into the given column. By assigning a value to `NEW.col_name` within the trigger, you cause the column to have that value in the new row.

This trigger is simple and its body contains only a single SQL statement. For a trigger body that executes multiple statements, use `BEGIN ... END` compound-statement syntax. In that case, if you use *mysql* to create the trigger, change the statement delimiter while you define the trigger, as discussed in Recipe 9.1.

To test the implementation, insert a row and check whether the trigger correctly initializes the sales tax rate for the invoice:

```
mysql> INSERT INTO cust_invoice (state,amount) VALUES('NY',100);
mysql> SELECT * FROM cust_invoice WHERE id = LAST_INSERT_ID();
+----+-------+--------+----------+
| id | state | amount | tax_rate |
+----+-------+--------+----------+
|  1 | NY    | 100.00 |     0.09 |
+----+-------+--------+----------+
```

The `SELECT` shows that the `tax_rate` column has the right value even though the `INSERT` provides no value for it.

9.5. Using Triggers to Simulate Function-Based Indexes

Problem

You need a function-based index, but MySQL doesn't support that capability.

Solution

Use a secondary column and triggers to simulate a function-based index.

Discussion

Some types of information are more easily analyzed using not the original values, but an expression computed from them. For example, if data values lie along an exponential curve, applying a logarithmic transform to them yields a more linear scale. Queries against a table that stores exponential values might therefore typically use expressions that refer to the log values:

```
SELECT * FROM expdata WHERE LOG10(value) < 2;
```

A disadvantage of such expressions is that referring to the `value` column within a function call prevents the optimizer from using any index on it. MySQL must retrieve the

values to apply the function to them, and the function values are not indexed. The result is diminished performance.

Some database systems permit an index to be defined on a function of a column, such that you can index `LOG10(value)`. MySQL does not support this capability, but there is a workaround:

1. Define a secondary column to store the function values and index that column.
2. Define triggers that keep the secondary column up to date when the original column is initialized or modified.
3. Refer directly to the secondary column in queries so that the optimizer can use the index on it for efficient lookups.

The following example illustrates this technique, using a table designed to store values that lie along an exponential curve:

```
CREATE TABLE expdata
(
  id          INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  value       FLOAT,  # original values
  log10_value FLOAT,  # LOG10() function of original values
  INDEX (value),      # index on original values
  INDEX (log10_value) # index on function-based values
);
```

The table includes `value` and `log10_value` columns to store the original data values and those values transformed with `LOG10()`. The table also indexes both columns.

Create an `INSERT` trigger to initialize the `log10_value` value from `value` for new rows, and an `UPDATE` trigger to keep `log10_value` up to date when `value` changes:

```
CREATE TRIGGER bi_expdata BEFORE INSERT ON expdata
FOR EACH ROW SET NEW.log10_value = LOG10(NEW.value);

CREATE TRIGGER bu_expdata BEFORE UPDATE ON expdata
FOR EACH ROW SET NEW.log10_value = LOG10(NEW.value);
```

To test the implementation, insert and modify some data and check the result of each operation:

```
mysql> INSERT INTO expdata (value) VALUES (.01),(.1),(1),(10),(100);
mysql> SELECT * FROM expdata;
+----+-------+-------------+
| id | value | log10_value |
+----+-------+-------------+
|  1 |  0.01 |          -2 |
|  2 |   0.1 |          -1 |
|  3 |     1 |           0 |
|  4 |    10 |           1 |
```

```
|  5 |   100 |           2 |
+----+-------+-------------+

mysql> UPDATE expdata SET value = value * 10;
mysql> SELECT * FROM expdata;
+----+-------+-------------+
| id | value | log10_value |
+----+-------+-------------+
|  1 |   0.1 |          -1 |
|  2 |     1 |           0 |
|  3 |    10 |           1 |
|  4 |   100 |           2 |
|  5 |  1000 |           3 |
+----+-------+-------------+
```

With that implementation, using a `log10_value` column that stores the `LOG10()` values of the `value` column, the `SELECT` query shown earlier can be rewritten:

```
SELECT * FROM expdata WHERE log10_value < 2;
```

The optimizer can use the index on `log10_value`, something not true of the original query that referred to `LOG10(value)`.

Using triggers this way to simulate a function-based index improves query performance for `SELECT` queries, but you should also consider the disadvantages of the technique:

- It requires extra storage for the secondary column.
- It requires more processing for statements that modify the original column (to activate the triggers that keep the secondary column and its index up to date).

The technique is therefore most useful if the workload for the table skews more toward retrievals than updates. It is less beneficial for a workload that is mostly updates.

The preceding example uses a `log10_value` column that is useful for several types of lookups, from single-row to range-based expressions. But functional indexes can be useful even for situations in which *most* queries select only a single row. Suppose that you want to store large data values such as PDF or XML documents in a table, but also want to look them up quickly later (for example, to access other values stored in the same row such as author or title). A `TEXT` or `BLOB` data type might be suitable for storing the values, but is not very suitable for finding them. (Comparisons in a lookup operation are slow for large values.) To work around this problem, use the following strategy:

1. Compute a hash value for each document and store it in the table along with the document. For example, use the `MD5()` function, which returns a 32-byte string of hexadecimal characters. That's still long for a comparison value, but much shorter than a full-column comparison based on contents of very long documents.
2. To look up the row containing a particular document, compute the document hash value and search the table for that value. For best performance, index the hash

column. Because the hash value is a function of the document, the index on it is, in effect, a functional index.

The result is that lookups based on the hash-value column will perform much more efficiently than lookups based on the original document values.

9.6. Simulating TIMESTAMP Properties for Other Date and Time Types

Problem

The `TIMESTAMP` data type provides auto-initialization and auto-update properties. You would like to use these properties for other temporal data types that permit only constant values for initialization and don't auto-update.

Solution

Use an `INSERT` trigger to provide the appropriate current date or time value at row-creation time. Use an `UPDATE` trigger to update the column to the current date or time when the row is changed.

Discussion

Recipe 6.7 describes the special `TIMESTAMP` and `DATETIME` initialization and update properties enable you to record row-creation and row-modification times automatically. These properties are not available for other temporal types, although there are reasons you might like them to be. For example, if you use separate `DATE` and `TIME` columns to store row-modification times, you can index the `DATE` column to enable efficient date-based lookups. (With `TIMESTAMP` or `DATETIME`, you cannot index just the date part of the column.)

One way to simulate `TIMESTAMP` properties for other temporal data types is to use the following strategy:

- When you create a row, initialize a `DATE` column to the current date and a `TIME` column to the current time.
- When you update a row, set the `DATE` and `TIME` columns to the new date and time.

However, this strategy requires all applications that use the table to implement the same strategy, and it fails if even one application neglects to do so. To place the burden of remembering to set the columns properly on the MySQL server and not on application writers, use triggers for the table. This is, in fact, a particular application of the general

strategy discussed in Recipe 9.4 that uses triggers to provide calculated values for initializing (or updating) row columns.

The following example shows how to use triggers to simulate TIMESTAMP properties for the DATE and TIME data types. (The same technique also serves to simulate TIMESTAMP properties for DATETIME for versions of MySQL older than 5.6.5, before DATETIME was given automatic properties.) Begin by creating the following table, which has a nontemporal column for storing data and columns for the DATE and TIME temporal types:

```
CREATE TABLE ts_emulate (data CHAR(10), d DATE, t TIME);
```

The intent here is that when applications insert or update values in the data column, MySQL should set the temporal columns appropriately to reflect the time at which modifications occur. To accomplish this, set up triggers that use the current date and time to initialize the temporal columns for new rows, and to update them when existing rows are changed. A BEFORE INSERT trigger handles row creation by invoking the CURDATE() and CURTIME() functions to get the current date and time and using those values to set the temporal columns:

```
CREATE TRIGGER bi_ts_emulate BEFORE INSERT ON ts_emulate
FOR EACH ROW SET NEW.d = CURDATE(), NEW.t = CURTIME();
```

A BEFORE UPDATE trigger handles updates to the temporal columns when the data column changes value. An IF statement is required here to emulate the TIMESTAMP property that an update occurs only if the data value in the row actually changes from its current value:

```
CREATE TRIGGER bu_ts_emulate BEFORE UPDATE ON ts_emulate
FOR EACH ROW # update temporal columns only if nontemporal column changes
IF NEW.data <> OLD.data THEN
  SET NEW.d = CURDATE(), NEW.t = CURTIME();
END IF;
```

To test the INSERT trigger, create a couple rows, but supply a value only for the data column. Then verify that MySQL provides the proper default values for the temporal columns:

```
mysql> INSERT INTO ts_emulate (data) VALUES('cat');
mysql> INSERT INTO ts_emulate (data) VALUES('dog');
mysql> SELECT * FROM ts_emulate;
+------+------------+----------+
| data | d          | t        |
+------+------------+----------+
| cat  | 2014-04-07 | 13:53:32 |
| dog  | 2014-04-07 | 13:53:37 |
+------+------------+----------+
```

Change the data value of one row to verify that the BEFORE UPDATE trigger updates the temporal columns of the changed row:

```
mysql> UPDATE ts_emulate SET data = 'axolotl' WHERE data = 'cat';
mysql> SELECT * FROM ts_emulate;
+---------+------------+----------+
| data    | d          | t        |
+---------+------------+----------+
| axolotl | 2014-04-07 | 13:53:49 |
| dog     | 2014-04-07 | 13:53:37 |
+---------+------------+----------+
```

Issue another `UPDATE`, but this time use one that does not change any `data` column values. In this case, the `BEFORE UPDATE` trigger should notice that no value change occurred and leave the temporal columns unchanged:

```
mysql> UPDATE ts_emulate SET data = data;
mysql> SELECT * FROM ts_emulate;
+---------+------------+----------+
| data    | d          | t        |
+---------+------------+----------+
| axolotl | 2014-04-07 | 13:53:49 |
| dog     | 2014-04-07 | 13:53:37 |
+---------+------------+----------+
```

The preceding example shows how to simulate the auto-initialization and auto-update properties offered by `TIMESTAMP` columns. To implement only one of those properties and not the other, create only one trigger and omit the other.

9.7. Using Triggers to Log Changes to a Table

Problem

You have a table that maintains current values of items that you track (such as auctions being bid on), but you'd also like to maintain a journal (history) of changes to the table.

Solution

Use triggers to "catch" table changes and write them to a separate log table.

Discussion

Suppose that you conduct online auctions, and that you maintain information about each currently active auction in a table that looks like this:

```
CREATE TABLE auction
(
  id   INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ts   TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  item VARCHAR(30) NOT NULL,
  bid  DECIMAL(10,2) NOT NULL,
```

```
  PRIMARY KEY (id)
);
```

The `auction` table contains information about the currently active auctions (items being bid on and the current bid for each auction). When an auction begins, insert a row into the table. For each bid on an item, update its `bid` column so that as the auction proceeds, the `ts` column updates to reflect the most recent bid time. When the auction ends, the `bid` value is the final price and the row can be removed from the table.

To maintain a journal that shows all changes to auctions as they progress from creation to removal, set up another table that serves to record a history of changes to the auctions. This strategy can be implemented with triggers.

To maintain a history of how each auction progresses, use an `auction_log` table with the following columns:

```
CREATE TABLE auction_log
(
  action ENUM('create','update','delete'),
  id     INT UNSIGNED NOT NULL,
  ts     TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  item   VARCHAR(30) NOT NULL,
  bid    DECIMAL(10,2) NOT NULL,
  INDEX (id)
);
```

The `auction_log` table differs from the `auction` table in two ways:

- It contains an `action` column to indicate for each row what kind of change was made.
- The `id` column has a nonunique index (rather than a primary key, which requires unique values). This permits multiple rows per `id` value because a given auction can generate many rows in the log table.

To ensure that changes to the `auction` table are logged to the `auction_log` table, create a set of triggers. The triggers write information to the `auction_log` table as follows:

- For inserts, log a row-creation operation showing the values in the new row.
- For updates, log a row-update operation showing the new values in the updated row.
- For deletes, log a row-removal operation showing the values in the deleted row.

For this application, `AFTER` triggers are used because they activate only after successful changes to the `auction` table. (`BEFORE` triggers might activate even if the row-change operation fails for some reason.) The trigger definitions look like this:

```
CREATE TRIGGER ai_auction AFTER INSERT ON auction
FOR EACH ROW
```

```
INSERT INTO auction_log (action,id,ts,item,bid)
VALUES('create',NEW.id,NOW(),NEW.item,NEW.bid);

CREATE TRIGGER au_auction AFTER UPDATE ON auction
FOR EACH ROW
INSERT INTO auction_log (action,id,ts,item,bid)
VALUES('update',NEW.id,NOW(),NEW.item,NEW.bid);

CREATE TRIGGER ad_auction AFTER DELETE ON auction
FOR EACH ROW
INSERT INTO auction_log (action,id,ts,item,bid)
VALUES('delete',OLD.id,OLD.ts,OLD.item,OLD.bid);
```

The `INSERT` and `UPDATE` triggers use `NEW.col_name` to access the new values being stored in rows. The `DELETE` trigger uses `OLD.col_name` to access the existing values from the deleted row. The `INSERT` and `UPDATE` triggers use `NOW()` to get the row-modification times; the `ts` column is initialized automatically to the current date and time, but `NEW.ts` will not contain that value.

Suppose that an auction is created with an initial bid of five dollars:

```
mysql> INSERT INTO auction (item,bid) VALUES('chintz pillows',5.00);
mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|              792 |
+------------------+
```

The `SELECT` statement fetches the auction ID value to use for subsequent actions on the auction. Then the item receives three more bids before the auction ends and is removed:

```
mysql> UPDATE auction SET bid = 7.50 WHERE id = 792;
... time passes ...
mysql> UPDATE auction SET bid = 9.00 WHERE id = 792;
... time passes ...
mysql> UPDATE auction SET bid = 10.00 WHERE id = 792;
... time passes ...
mysql> DELETE FROM auction WHERE id = 792;
```

At this point, no trace of the auction remains in the `auction` table, but the `auction_log` table contains a complete history of what occurred:

```
mysql> SELECT * FROM auction_log WHERE id = 792 ORDER BY ts;
+--------+-----+---------------------+---------------+-------+
| action | id  | ts                  | item          | bid   |
+--------+-----+---------------------+---------------+-------+
| create | 792 | 2014-01-09 14:57:41 | chintz pillows |  5.00 |
| update | 792 | 2014-01-09 14:57:50 | chintz pillows |  7.50 |
| update | 792 | 2014-01-09 14:57:57 | chintz pillows |  9.00 |
| update | 792 | 2014-01-09 14:58:03 | chintz pillows | 10.00 |
| delete | 792 | 2014-01-09 14:58:03 | chintz pillows | 10.00 |
+--------+-----+---------------------+---------------+-------+
```

With the strategy just outlined, the `auction` table remains relatively small, and you can always find information about auction histories as necessary by looking in the `auction_log` table.

9.8. Using Events to Schedule Database Actions

Problem

You want to set up a database operation that runs periodically without user intervention.

Solution

Create an event that executes according to a schedule.

Discussion

MySQL provides an event scheduler that enables you to set up database operations that run at times that you define. This section describes what you must do to use events, beginning with a simple event that writes a row to a table at regular intervals. Why bother creating such an event? One reason is that the rows serve as a log of continuous server operation, similar to the `MARK` line that some Unix `syslogd` servers write to the system log periodically so that you know they're alive.

Begin with a table to hold the mark rows. It contains a `TIMESTAMP` column (which MySQL will initialize automatically) and a column to store a message:

```
CREATE TABLE mark_log
(
  ts      TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  message VARCHAR(100)
);
```

Our logging event will write a string to a new row. To set it up, use a `CREATE EVENT` statement:

```
CREATE EVENT mark_insert
ON SCHEDULE EVERY 5 MINUTE
DO INSERT INTO mark_log (message) VALUES('-- MARK --');
```

The `mark_insert` event causes the message `'-- MARK --'` to be logged to the `mark_log` table every five minutes. Use a different interval for more or less frequent logging.

This event is simple and its body contains only a single SQL statement. For an event body that executes multiple statements, use `BEGIN` … `END` compound-statement syntax. In that case, if you use *mysql* to create the event, change the statement delimiter while you define the event, as discussed in Recipe 9.1.

At this point, you should wait a few minutes and then select the contents of the `mark_log` table to verify that new rows are being written on schedule. However, if this is the first event that you've set up, you might find that the table remains empty no matter how long you wait:

```
mysql> SELECT * FROM mark_log;
Empty set (0.00 sec)
```

If that's the case, very likely the event scheduler isn't running (which is its default state until you enable it). Check the scheduler status by examining the value of the `event_scheduler` system variable:

```
mysql> SHOW VARIABLES LIKE 'event_scheduler';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| event_scheduler | OFF   |
+-----------------+-------+
```

To enable the scheduler interactively if it's not running, execute the following statement (which requires the `SUPER` privilege):

```
SET GLOBAL event_scheduler = 1;
```

That statement enables the scheduler, but only until the server shuts down. To start the scheduler each time the server starts, enable the system variable in your *my.cnf* option file:

```
[mysqld]
event_scheduler=1
```

When the event scheduler is enabled, the `mark_insert` event eventually creates many rows in the table. There are several ways that you can affect event execution to prevent the table from growing forever:

- Drop the event:

  ```
  DROP EVENT mark_insert;
  ```

  This is the simplest way to stop an event from occurring. But if you want it to resume later, you must re-create it.

- Disable event execution:

  ```
  ALTER EVENT mark_insert DISABLE;
  ```

  That leaves the event in place but causes it not to run until you reactivate it:

  ```
  ALTER EVENT mark_insert ENABLE;
  ```

- Let the event continue to run, but set up another event that "expires" old `mark_log` rows. This second event need not run so frequently (perhaps once a day). Its body should remove rows older than a given threshold. The following definition creates an event that deletes rows that are more than two days old:

```
CREATE EVENT mark_expire
ON SCHEDULE EVERY 1 DAY
DO DELETE FROM mark_log WHERE ts < NOW() - INTERVAL 2 DAY;
```

If you adopt this strategy, you have cooperating events: one event that adds rows to the `mark_log` table, and another that removes them. They act together to maintain a log that contains recent rows but does not become too large.

9.9. Writing Helper Routines for Executing Dynamic SQL

Problem

Prepared SQL statements enable you to construct and execute SQL statements on the fly, but the supporting mechanism can be tedious to use.

Solution

Write a helper procedure that handles the drudgery.

Discussion

Using a prepared SQL statement involves three steps: preparation, execution, and deallocation. For example, if the `@tbl_name` and `@val` variables hold a table name and a value to insert into the table, you can create the table and insert the value like this:

```
SET @stmt = CONCAT('CREATE TABLE ',@tbl_name,' (i INT)');
PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
SET @stmt = CONCAT('INSERT INTO ',@tbl_name,' (i) VALUES(',@val,')');
PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

To ease the burden of going through those steps for each dynamically created statement, use a helper routine that, given a statement string, prepares, executes, and deallocates it:

```
CREATE PROCEDURE exec_stmt(stmt_str TEXT)
BEGIN
  SET @_stmt_str = stmt_str;
  PREPARE stmt FROM @_stmt_str;
  EXECUTE stmt;
  DEALLOCATE PREPARE stmt;
END;
```

The `exec_stmt()` routine enables the same statements to be executed much more simply:

```
CALL exec_stmt(CONCAT('CREATE TABLE ',@tbl_name,' (i INT)'));
CALL exec_stmt(CONCAT('INSERT INTO ',@tbl_name,' (i) VALUES(',@val,')'));
```

`exec_stmt()` uses an intermediary user-defined variable, `@_exec_stmt`, because `PREPARE` accepts a statement only when specified using either a literal string or a user-defined variable. A statement stored in a routine parameter does not work. (Avoid using `@_exec_stmt` for your own purposes, at least if you expect its value to persist across `exec_stmt()` invocations.)

Now, how about making it safer to construct statement strings that incorporate values that might come from external sources, such as web-form input or command-line arguments? Such information cannot be trusted and should be treated as a potential SQL injection attack vector:

- The `QUOTE()` function is available for quoting data values.
- There is no corresponding function for identifiers, but it's easy to write one that doubles internal backticks and adds a backtick at the beginning and end:

```
CREATE FUNCTION quote_identifier(id TEXT)
RETURNS TEXT DETERMINISTIC
RETURN CONCAT('`',REPLACE(id,'`','``'),'`');
```

Revising the preceding example to ensure the safety of data values and identifiers, we have:

```
SET @tbl_name = quote_identifier(@tbl_name);
SET @val = QUOTE(@val);
CALL exec_stmt(CONCAT('CREATE TABLE ',@tbl_name,' (i INT)'));
CALL exec_stmt(CONCAT('INSERT INTO ',@tbl_name,' (i) VALUES(',@val,')'));
```

A constraint on use of `exec_stmt()` is that not all SQL statements are eligible for execution as prepared statements. See the *MySQL Reference Manual* for the limitations.

9.10. Handling Errors Within Stored Programs

Within stored programs, you can catch errors or exceptional conditions using condition handlers. A handler activates under specific circumstances, causing the code associated with it to execute. The code takes suitable action such as performing cleanup processing or setting a variable that can be tested elsewhere in the program to determine whether the condition occurred. A handler might even ignore an error if it occurs under certain permitted conditions and you want to catch it rather than have it terminate your program.

Stored programs can also produce their own errors or warnings to signal that something has gone wrong.

The following examples illustrate these techniques. For complete lists of available condition names, SQLSTATE values, and error codes, consult the *MySQL Reference Manual*.

Detecting End-of-Data Conditions

One common use of condition handlers is to detect "no more rows" conditions. To process a query result one row at a time, use a cursor-based fetch loop in conjunction with a condition handler that catches the end-of-data condition. The technique has these essential elements:

- A cursor associated with a `SELECT` statement that reads rows. Open the cursor to start reading, and close it to stop.
- A condition handler that activates when the cursor reaches the end of the result set and raises an end-of-data condition (`NOT FOUND`). We used a similar handler in Recipe 9.2.
- A variable that indicates loop termination. Initialize the variable to `FALSE`, then set it to `TRUE` within the condition handler when the end-of-data condition occurs.
- A loop that uses the cursor to fetch each row and exits when the loop-termination variable becomes `TRUE`.

The following example implements a fetch loop that processes the `states` table row by row to calculate the total US population:

```
CREATE PROCEDURE us_population()
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE state_pop, total_pop BIGINT DEFAULT 0;
  DECLARE cur CURSOR FOR SELECT pop FROM states;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur;
  fetch_loop: LOOP
    FETCH cur INTO state_pop;
    IF done THEN
      LEAVE fetch_loop;
    END IF;
    SET total_pop = total_pop + state_pop;
  END LOOP;
  CLOSE cur;
  SELECT total_pop AS 'Total U.S. Population';
END;
```

Clearly, that example is purely for illustration because in any real application you'd use an aggregate function to calculate the total. But that also gives us an independent check on whether the fetch loop calculates the correct value:

```
mysql> CALL us_population();
+----------------------+
| Total U.S. Population |
+----------------------+
|             308143815 |
+----------------------+
mysql> SELECT SUM(pop) AS 'Total U.S. Population' FROM states;
+----------------------+
| Total U.S. Population |
+----------------------+
|             308143815 |
+----------------------+
```

`NOT FOUND` handlers are also useful for checking whether `SELECT … INTO` *`var_name`* statements return any results. Recipe 9.2 shows an example.

Catching and Ignoring Errors

If you consider an error benign, you can use a handler to ignore it. For example, many `DROP` statements in MySQL have an `IF EXISTS` clause to suppress errors if objects to be dropped do not exist. But some `DROP` statements have no such clause and thus no way to suppress errors. `DROP USER` is one of these:

```
mysql> DROP USER 'bad-user'@'localhost';
ERROR 1396 (HY000): Operation DROP USER failed for 'bad-user'@'localhost'
```

To prevent errors from occurring for nonexistent users, invoke `DROP USER` within a stored procedure that catches code 1396 and ignores it:

```
CREATE PROCEDURE drop_user(user TEXT, host TEXT)
BEGIN
  DECLARE account TEXT;
  DECLARE CONTINUE HANDLER FOR 1396
    SELECT CONCAT('Unknown user: ', account) AS Message;
  SET account = CONCAT(QUOTE(user),'@',QUOTE(host));
  CALL exec_stmt(CONCAT('DROP USER ',account));
END;
```

If the user does not exist, `drop_user()` writes a message within the condition handler, but no error occurs:

```
mysql> CALL drop_user('bad-user','localhost');
+-------------------------------------+
| Message                             |
+-------------------------------------+
| Unknown user: 'bad-user'@'localhost' |
+-------------------------------------+
```

To ignore the error completely, write the handler using an empty `BEGIN … END` block:

```
DECLARE CONTINUE HANDLER FOR 1396 BEGIN END;
```

Another approach is to generate a warning, as demonstrated in the next section.

Raising Errors and Warnings

To produce your own errors within a stored program when you detect something awry, use the SIGNAL statement. This section shows some examples, and Recipe 9.11 demonstrates use of SIGNAL within a trigger to reject bad data.

Suppose that an application performs a division operation for which you expect that the divisor will never be zero, and that you want to produce an error otherwise. You might expect that you could set the SQL mode properly to produce a divide-by-zero error (this requires ERROR_FOR_DIVISION_BY_ZERO plus strict mode, or just strict mode as of MySQL 5.7.4). But that works only within the context of data-modification operations such as INSERT. In other contexts, division by zero produces only a warning:

```
mysql> SET sql_mode = 'ERROR_FOR_DIVISION_BY_ZERO,STRICT_ALL_TABLES';
mysql> SELECT 1/0;
+------+
| 1/0  |
+------+
| NULL |
+------+
1 row in set, 1 warning (0.00 sec)
mysql> SHOW WARNINGS;
+---------+------+---------------+
| Level   | Code | Message       |
+---------+------+---------------+
| Warning | 1365 | Division by 0 |
+---------+------+---------------+
```

To ensure a divide-by-zero error in any context, write a function that performs the division but checks the divisor first and uses SIGNAL to raise an error if the "can't happen" condition occurs:

```
CREATE FUNCTION divide(numerator FLOAT, divisor FLOAT)
RETURNS FLOAT DETERMINISTIC
BEGIN
  IF divisor = 0 THEN
    SIGNAL SQLSTATE '22012'
           SET MYSQL_ERRNO = 1365, MESSAGE_TEXT = 'unexpected 0 divisor';
  END IF;
  RETURN numerator / divisor;
END;
```

Test the function in a nonmodification context to verify that it produces an error:

```
mysql> SELECT divide(1,0);
ERROR 1365 (22012): unexpected 0 divisor
```

The SIGNAL statement specifies a SQLSTATE value plus an optional SET clause you can use to assign values to error attributes. MYSQL_ERRNO corresponds to the MySQL-specific error code, and MESSAGE_TEXT is a string of your choice.

SIGNAL can also raise warning conditions, not just errors. The following routine, drop_user_warn(), is similar to the drop_user() routine shown earlier, but instead of printing a message for nonexistent users, it generates a warning that can be displayed with SHOW WARNINGS. SQLSTATE value 01000 and error 1642 indicate a user-defined unhandled exception, which the routine signals along with an appropriate message:

```
CREATE PROCEDURE drop_user_warn(user TEXT, host TEXT)
BEGIN
  DECLARE account TEXT;
  DECLARE CONTINUE HANDLER FOR 1396
  BEGIN
    DECLARE msg TEXT;
    SET msg = CONCAT('Unknown user: ', account);
    SIGNAL SQLSTATE '01000' SET MYSQL_ERRNO = 1642, MESSAGE_TEXT = msg;
  END;
  SET account = CONCAT(QUOTE(user),'@',QUOTE(host));
  CALL exec_stmt(CONCAT('DROP USER ',account));
END;
```

Give it a test:

```
mysql> CALL drop_user_warn('bad-user','localhost');
Query OK, 0 rows affected, 1 warning (0.00 sec)
mysql> SHOW WARNINGS;
+---------+------+--------------------------------------+
| Level   | Code | Message                              |
+---------+------+--------------------------------------+
| Warning | 1642 | Unknown user: 'bad-user'@'localhost' |
+---------+------+--------------------------------------+
```

9.11. Using Triggers to Preprocess or Reject Data

Problem

There are conditions you want to check for data entered into a table, but you don't want to write the validation logic for every INSERT.

Solution

Centralize the input-testing logic into a BEFORE INSERT trigger.

Discussion

You can use triggers to perform several types of input checks:

- Reject bad data by raising a signal. This gives you access to stored program logic for more latitude in checking values than is possible with static constraints such as NOT NULL.

- Preprocess values and modify them, if you won't want to reject them outright. For example, map out-of-range values to be in range or sanitize values to put them in canonical form, if you permit entry of close variants.

Suppose that you have a table of contact information such as name, state of residence, email address, and website URL:

```
CREATE TABLE contact_info
(
  id     INT NOT NULL AUTO_INCREMENT,
  name   VARCHAR(30),   # state of residence
  state  CHAR(2),       # state of residence
  email  VARCHAR(50),   # email address
  url    VARCHAR(255),  # web address
  PRIMARY KEY (id)
);
```

For entry of new rows, you want to enforce constraints or perform preprocessing as follows:

- State of residence values are two-letter US state codes, valid only if present in the `states` table. (In this case, you could declare the column as an `ENUM` with 50 members, so it's more likely you'd use this lookup-table technique with columns for which the set of valid values is arbitrarily large or changes over time.)
- Email address values must contain an `@` character to be valid.
- For website URLs, strip any leading `http://` to save space.

To handle these requirements, create a `BEFORE INSERT` trigger:

```
CREATE TRIGGER bi_contact_info BEFORE INSERT ON contact_info
FOR EACH ROW
BEGIN
  IF (SELECT COUNT(*) FROM states WHERE abbrev = NEW.state) = 0 THEN
    SIGNAL SQLSTATE 'HY000'
           SET MYSQL_ERRNO = 1525, MESSAGE_TEXT = 'invalid state code';
  END IF;
  IF INSTR(NEW.email,'@') = 0 THEN
    SIGNAL SQLSTATE 'HY000'
           SET MYSQL_ERRNO = 1525, MESSAGE_TEXT = 'invalid email address';
  END IF;
  SET NEW.url = TRIM(LEADING 'http://' FROM NEW.url);
END;
```

To also handle updates, define a `BEFORE UPDATE` trigger with the same body as `bi_contact_info`.

Test the trigger by executing some `INSERT` statements to verify that it accepts valid values, rejects bad ones, and trims URLs:

```
mysql> INSERT INTO contact_info (name,state,email,url)
    -> VALUES('Jen','NY','jen@example.com','http://www.example.com');
mysql> INSERT INTO contact_info (name,state,email,url)
    -> VALUES('Jen','XX','jen@example.com','http://www.example.com');
ERROR 1525 (HY000): invalid state code
mysql> INSERT INTO contact_info (name,state,email,url)
    -> VALUES('Jen','NY','jen','http://www.example.com');
ERROR 1525 (HY000): invalid email address
mysql> SELECT * FROM contact_info;
+----+------+-------+-----------------+-----------------+
| id | name | state | email           | url             |
+----+------+-------+-----------------+-----------------+
|  1 | Jen  | NY    | jen@example.com | www.example.com |
+----+------+-------+-----------------+-----------------+
```

CHAPTER 10
Working with Metadata

10.0. Introduction

Most of the SQL statements used so far have been written to work with the data stored in the database. That is, after all, what the database is designed to hold. But sometimes you need more than just data values. You need information that characterizes or describes those values—that is, the statement metadata. Metadata is used most often to process result sets, but also applies to other aspects of your interaction with MySQL. This chapter describes how to obtain and use several types of metadata:

Information about statement results
: For statements that delete or update rows, you can determine how many rows were changed. For a `SELECT` statement, you can obtain the number of columns in the result set, as well as information about each column in the result set, such as the column name and its display width. For example, to format a tabular display, you can determine how wide to make each column and whether to justify values to the left or right.

Information about databases and tables
: A MySQL server can be queried to determine which databases and tables it manages, which is useful for existence tests or producing lists. For example, an application might present a display enabling the user to select one of the available databases. Table metadata can be examined to determine column definitions; for example, to determine the legal values for `ENUM` or `SET` columns to generate web form elements corresponding to the available choices.

Information about the MySQL server
: The database server provides information about itself and about the status of your current session with it. Knowing the server version can be useful for determining whether it supports a given feature, which helps you build adaptive applications.

Metadata is closely tied to the implementation of the database system, so it tends to be database system–dependent. This means that if an application uses techniques shown in this chapter, it might need some modification if you port it to other database systems. For example, lists of tables and databases in MySQL are available by executing `SHOW` statements. However, `SHOW` is a MySQL-specific extension to SQL, so even for APIs like Perl or Ruby DBI, PDO, DB API, and JDBC that give you a database-independent way of executing statements, the SQL itself is MySQL-specific and must be changed to work with other database systems.

A more portable source of metadata is `INFORMATION_SCHEMA`, a database that contains information about databases, tables, columns, character sets, and so forth. `INFORMATION_SCHEMA` has some advantages over `SHOW`:

- Other database systems support `INFORMATION_SCHEMA`, so applications that use it are likely to be more portable than those that use `SHOW` statements.
- `INFORMATION_SCHEMA` is used with standard `SELECT` syntax, so it's more similar to other data-retrieval operations than `SHOW` statements.

Because of those advantages, recipes in this chapter use `INFORMATION_SCHEMA` rather than `SHOW` in most cases.

A disadvantage of `INFORMATION_SCHEMA` is that statements to access it are more verbose than the corresponding `SHOW` statements. That doesn't matter so much when you're writing programs, but for interactive use, `SHOW` statements can be more attractive because they require less typing. The following table lists `SHOW` statements that provide information similar to the contents of certain `INFORMATION_SCHEMA` tables:

INFORMATION_SCHEMA table	SHOW statement
SCHEMATA	SHOW DATABASES
TABLES	SHOW TABLES
COLUMNS	SHOW COLUMNS

The results retrieved from `INFORMATION_SCHEMA` or `SHOW` depend on your privileges. You'll see information only for those databases or tables for which you have some privileges. Thus, an existence test for an object returns false if it exists but you have no privileges for accessing it.

Scripts that create tables used in this chapter are located in the *tables* directory of the `recipes` distribution. Scripts containing code for the examples are located in the *metadata* directory. (Some of them use utility functions located in the *lib* directory.) The distribution often provides implementations in languages other than those shown.

10.1. Determining the Number of Rows Affected by a Statement

Problem

You want to know how many rows an SQL statement changed.

Solution

Sometimes the row count is the return value of the function that executes the statement. Other times the count is returned by a separate function that you call after executing the statement.

Discussion

For statements that affect rows (`UPDATE`, `DELETE`, `INSERT`, `REPLACE`), each API provides a way to determine the number of rows involved. For MySQL, the default meaning of "affected by" is "changed by," not "matched by." That is, rows not changed by a statement are not counted, even if they match the conditions specified in the statement. For example, the following `UPDATE` statement results in an "affected by" value of zero because it changes no columns from their current values, no matter how many rows the `WHERE` clause matches:

```
UPDATE profile SET cats = 0 WHERE cats = 0;
```

The MySQL server permits a client to set a connect-time flag to indicate that it wants rows-matched counts, not rows-changed counts. In this case, the row count for the preceding statement would be equal to the number of rows with an `arms` value of 0, even though the statement results in no net change to the table. However, not all MySQL APIs expose this flag. The following discussion indicates which APIs enable you to select the type of count you want and which use the rows-matched count by default rather than the rows-changed count.

Perl

In Perl DBI scripts, `do()` returns the row count for statements that modify rows:

```
my $count = $dbh->do ($stmt);
# report 0 rows if an error occurred
printf "Number of rows affected: %d\n", (defined ($count) ? $count : 0);
```

If you prepare a statement first and then execute it, `execute()` returns the row count:

```
my $sth = $dbh->prepare ($stmt);
my $count = $sth->execute ();
printf "Number of rows affected: %d\n", (defined ($count) ? $count : 0);
```

To tell MySQL whether to return rows-changed or rows-matched counts, specify `mysql_client_found_rows` in the options part of the data source name (DSN) argument of the `connect()` call when you connect to the MySQL server. Set the option to 0 for rows-changed counts and 1 for rows-matched counts. Here's an example:

```
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
my $dsn = "DBI:mysql:cookbook:localhost;mysql_client_found_rows=1";
my $dbh = DBI->connect ($dsn, "cbuser", "cbpass", $conn_attrs);
```

`mysql_client_found_rows` changes the row-reporting behavior for the duration of the session.

Although the default behavior for MySQL itself is to return rows-changed counts, current versions of the Perl DBI driver for MySQL automatically request rows-matched counts unless you specify otherwise. For applications that depend on a particular behavior, it's best to explicitly set the `mysql_client_found_rows` option in the DSN to the appropriate value.

Ruby

In Ruby DBI scripts, the `do` method returns the row count for statements that modify rows:

```
count = dbh.do(stmt)
puts "Number of rows affected: #{count}"
```

If you use `execute` to execute a statement, use the statement handle `rows` method to get the count afterward:

```
sth = dbh.execute(stmt)
puts "Number of rows affected: #{sth.rows}"
```

The Ruby DBI driver for MySQL returns rows-changed counts by default, but the driver supports a `mysql_client_found_rows` option that enables you to control whether the server returns rows-changed or rows-matched counts. Its use is analogous to Perl DBI. For example, to request rows-matched counts, do this:

```
dsn = "DBI:Mysql:database=cookbook;host=localhost;mysql_client_found_rows=1"
dbh = DBI.connect(dsn, "cbuser", "cbpass")
```

PHP

In PDO, the database handle `exec()` method returns the rows-affected count:

```
$count = $dbh->exec ($stmt);
printf ("Number of rows updated: %d\n", $count);
```

If you use `prepare()` plus `execute()` instead, the rows-affected count is available from the statement handle `rowCount()` method:

```
$sth = $dbh->prepare ($stmt);
$sth->execute ();
printf ("Number of rows updated: %d\n", $sth->rowCount ());
```

The PDO driver for MySQL returns rows-changed counts by default, but the driver supports a `PDO::MYSQL_ATTR_FOUND_ROWS` attribute that you can specify at connect time to control whether the server returns rows-changed or rows-matched counts. The `new PDO` class constructor takes an optional key/value array following the password argument. Pass `PDO::MYSQL_ATTR_FOUND_ROWS => 1` in this array to request rows-matched counts:

```
$dsn = "mysql:host=localhost;dbname=cookbook";
$dbh = new PDO ($dsn, "cbuser", "cbpass",
                array (PDO::MYSQL_ATTR_FOUND_ROWS => 1));
```

Python

Python's DB API makes the rows-changed count available as the value of the statement cursor's `rowcount` attribute:

```
cursor = conn.cursor()
cursor.execute(stmt)
print("Number of rows affected: %d" % cursor.rowcount)
cursor.close()
```

To obtain rows-matched counts instead, import the Connector/Python client-flag constants and pass the `FOUND_ROWS` flag in the `client_flags` parameter of the `connect()` method:

```
from mysql.connector.constants import ClientFlag

conn = mysql.connector.connect(
  database="cookbook",
  host="localhost",
  user="cbuser",
  password="cbpass",
  client_flags=[ClientFlag.FOUND_ROWS]
)
```

Java

For statements that modify rows, the Connector/J driver provides rows-matched counts rather than rows-changed counts, for conformance with the Java JDBC specification.

The JDBC interface provides row counts two different ways, depending on the method you invoke to execute the statement. If you use `executeUpdate()`, the row count is its return value:

```
Statement s = conn.createStatement ();
int count = s.executeUpdate (stmt);
s.close ();
System.out.println ("Number of rows affected: " + count);
```

If you use `execute()`, that method returns true or false to indicate whether the statement produces a result set. For statements such as `UPDATE` or `DELETE` that return no result set, `execute()` returns false and the row count is available by calling the `getUpdateCount()` method:

```
Statement s = conn.createStatement ();
if (!s.execute (stmt))
{
  // there is no result set, print the row count
  System.out.println ("Number of rows affected: " + s.getUpdateCount ());
}
s.close ();
```

10.2. Obtaining Result Set Metadata

Problem

You already know how to retrieve the rows of a result set (see Recipe 2.4). Now you want to know things *about* the result set, such as the column names and data types, or how many rows and columns there are.

Solution

Use the capabilities provided by your API.

Discussion

Statements such as `SELECT` that generate a result set produce several types of metadata. This section discusses the information available through each API, using programs that show how to display the result set metadata available after executing a sample statement (`SELECT name, birth FROM profile`). The example programs illustrate one of the simplest uses for this information: when you retrieve a row from a result set and you want to process the column values in a loop, the column count stored in the metadata serves as the upper bound on the loop iterator.

Perl

The scope of result set metadata available from Perl DBI depends on how you process queries:

- Using a statement handle

  In this case, invoke `prepare()` to get the statement handle. This handle has an `execute()` method. Invoke it to generate the result set, then fetch the rows in a loop. With this approach, access to the metadata is available while the result set is active—that is, after the call to `execute()` and until the end of the result set is reached.

When the row-fetching method finds that there are no more rows, it invokes `finish()` implicitly, which causes the metadata to become unavailable. (That also happens if you explicitly call `finish()` yourself.) Thus, normally it's best to access the metadata immediately after calling `execute()`, making a copy of any values that you'll need to use beyond the end of the fetch loop.

- Using a database-handle method that returns the result set in a single operation

  With this approach, any metadata generated while processing the statement will have been disposed of by the time the method returns. You can still determine the number of rows and columns from the size of the result set.

When you use a statement handle to process a query, DBI makes result set metadata available after you invoke the handle's `execute()` method. This information is available primarily in the form of references to arrays. For each such type of metadata, the array has one element per column in the result set. Access these array references as attributes of the statement handle. For example, `$sth->{NAME}` points to the column name array, with individual column names available as elements of this array:

```
$name = $sth->{NAME}->[$i];
```

Or access the entire array like this:

```
@names = @{$sth->{NAME}};
```

The following table lists the attribute names through which you access array-based metadata and the meaning of values in each array. Names that begin with uppercase are standard DBI attributes and should be available for most database engines. Attribute names that begin with `mysql_` are MySQL-specific and nonportable:

Attribute name	Array element meaning
`NAME`	Column name
`NAME_lc`	Column name in lowercase
`NAME_uc`	Column name in uppercase
`NULLABLE`	0 or empty string = column values cannot be `NULL`
	1 = column values can be `NULL`
	2 = unknown
`PRECISION`	Column width
`SCALE`	Number of decimal places (for numeric columns)
`TYPE`	Data type (numeric DBI code)
`mysql_is_blob`	True if column has a `BLOB` (or `TEXT`) type
`mysql_is_key`	True if column is part of a key
`mysql_is_num`	True if column has a numeric type
`mysql_is_pri_key`	True if column is part of a primary key
`mysql_max_length`	Actual maximum length of column values in result set

Attribute name	Array element meaning
mysql_table	Name of table the column is part of
mysql_type	Data type (numeric internal MySQL code)
mysql_type_name	Data type name

Some types of metadata, listed in the following table, are accessed as references to hashes rather than arrays. These hashes have one element per column value. The element key is the column name and its value is the position of the column within the result set. For example:

```
$col_pos = $sth->{NAME_hash}->{col_name};
```

Attribute name	Hash element meaning
NAME_hash	Column name
NAME_hash_lc	Column name in lowercase
NAME_hash_uc	Column name in uppercase

The number of columns in a result set is available as a scalar value:

```
$num_cols = $sth->{NUM_OF_FIELDS};
```

This example code shows how to execute a statement and display result set metadata:

```
my $stmt = "SELECT name, birth FROM profile";
printf "Statement: %s\n", $stmt;
my $sth = $dbh->prepare ($stmt);
$sth->execute();
# metadata information becomes available at this point ...
printf "NUM_OF_FIELDS: %d\n", $sth->{NUM_OF_FIELDS};
print "Note: statement has no result set\n" if $sth->{NUM_OF_FIELDS} == 0;
for my $i (0 .. $sth->{NUM_OF_FIELDS}-1)
{
  printf "--- Column %d (%s) ---\n", $i, $sth->{NAME}->[$i];
  printf "NAME_lc:          %s\n", $sth->{NAME_lc}->[$i];
  printf "NAME_uc:          %s\n", $sth->{NAME_uc}->[$i];
  printf "NULLABLE:         %s\n", $sth->{NULLABLE}->[$i];
  printf "PRECISION:        %d\n", $sth->{PRECISION}->[$i];
  printf "SCALE:            %d\n", $sth->{SCALE}->[$i];
  printf "TYPE:             %d\n", $sth->{TYPE}->[$i];
  printf "mysql_is_blob:    %s\n", $sth->{mysql_is_blob}->[$i];
  printf "mysql_is_key:     %s\n", $sth->{mysql_is_key}->[$i];
  printf "mysql_is_num:     %s\n", $sth->{mysql_is_num}->[$i];
  printf "mysql_is_pri_key: %s\n", $sth->{mysql_is_pri_key}->[$i];
  printf "mysql_max_length: %d\n", $sth->{mysql_max_length}->[$i];
  printf "mysql_table:      %s\n", $sth->{mysql_table}->[$i];
  printf "mysql_type:       %d\n", $sth->{mysql_type}->[$i];
  printf "mysql_type_name:  %s\n", $sth->{mysql_type_name}->[$i];
}
$sth->finish ();  # release result set because we didn't fetch its rows
```

The program produces this output:

```
Statement: SELECT name, birth FROM profile
NUM_OF_FIELDS: 2
--- Column 0 (name) ---
NAME_lc:          name
NAME_uc:          NAME
NULLABLE:
PRECISION:        20
SCALE:            0
TYPE:             12
mysql_is_blob:
mysql_is_key:
mysql_is_num:     0
mysql_is_pri_key:
mysql_max_length: 7
mysql_table:      profile
mysql_type:       253
mysql_type_name:  varchar
--- Column 1 (birth) ---
NAME_lc:          birth
NAME_uc:          BIRTH
NULLABLE:         1
PRECISION:        10
SCALE:            0
TYPE:             9
mysql_is_blob:
mysql_is_key:
mysql_is_num:     0
mysql_is_pri_key:
mysql_max_length: 10
mysql_table:      profile
mysql_type:       10
mysql_type_name:  date
```

To get a row count from a result set generated by calling `execute()`, fetch the rows and count them yourself. Using `$sth->rows()` to get a count for `SELECT` statements is expressly deprecated in the DBI documentation.

You can also obtain a result set by calling one of the DBI methods that uses a database handle rather than a statement handle, such as `selectall_arrayref()` or `selectall_hashref()`. These methods provide no access to column metadata. That information already will have been disposed of by the time the method returns, and is unavailable to your scripts. However, you can derive column and row counts by examining the result set itself. Recipe 2.4 discusses the result set structures produced by several methods and how to use them to obtain row and column counts.

Ruby

Ruby DBI provides result set metadata after you execute a statement with `execute`, and access to metadata is possible until you invoke the statement handle `finish` method.

The `column_names` method returns an array of column names (which is empty if there is no result set). If there is a result set, the `column_info` method returns an array of `ColumnInfo` objects, one for each column. A `ColumnInfo` object is similar to a hash and has the elements shown in the following table. Element names that begin with `mysql_` are MySQL-specific and nonportable:

Element name	Element meaning
`name`	Column name
`sql_type`	XOPEN type number
`type_name`	XOPEN type name
`precision`	Column width
`scale`	Number of decimal places (for numeric columns)
`nullable`	True if column permits `NULL` values
`indexed`	True if column is indexed
`primary`	True if column is part of a primary key
`unique`	True if column is part of a unique index
`mysql_type`	Data type (numeric internal MySQL code)
`mysql_type_name`	Data type name
`mysql_length`	Column width
`mysql_max_length`	Actual maximum length of column values in result set
`mysql_flags`	Data type flags

This example code shows how to execute a statement and display result set metadata:

```
stmt = "SELECT name, birth FROM profile"
puts "Statement: #{stmt}"
sth = dbh.execute(stmt)
# metadata information becomes available at this point ...
puts "Number of columns: #{sth.column_names.size}"
puts "Note: statement has no result set" if sth.column_names.size == 0
sth.column_info.each_with_index do |info, i|
  puts "--- Column #{i} (#{info['name']}) ---"
  puts "sql_type:         #{info['sql_type']}"
  puts "type_name:        #{info['type_name']}"
  puts "precision:        #{info['precision']}"
  puts "scale:            #{info['scale']}"
  puts "nullable:         #{info['nullable']}"
  puts "indexed:          #{info['indexed']}"
  puts "primary:          #{info['primary']}"
  puts "unique:           #{info['unique']}"
  puts "mysql_type:       #{info['mysql_type']}"
  puts "mysql_type_name:  #{info['mysql_type_name']}"
  puts "mysql_length:     #{info['mysql_length']}"
  puts "mysql_max_length: #{info['mysql_max_length']}"
  puts "mysql_flags:      #{info['mysql_flags']}"
```

```
end
sth.finish
```

The program produces this output:

```
Statement: SELECT name, birth FROM profile
Number of columns: 2
--- Column 0 (name) ---
sql_type:         12
type_name:        VARCHAR
precision:        20
scale:            0
nullable:         false
indexed:          false
primary:          false
unique:           false
mysql_type:       253
mysql_type_name:  VARCHAR
mysql_length:     20
mysql_max_length: 7
mysql_flags:      4097
--- Column 1 (birth) ---
sql_type:         9
type_name:        DATE
precision:        10
scale:            0
nullable:         true
indexed:          false
primary:          false
unique:           false
mysql_type:       10
mysql_type_name:  DATE
mysql_length:     10
mysql_max_length: 10
mysql_flags:      128
```

To get a row count from a result set generated by calling `execute`, fetch the rows and count them yourself. The `sth.rows` method is not guaranteed to work for result sets.

You can also obtain a result set by calling one of the DBI methods that uses a database handle rather than a statement handle, such as `select_one` or `select_all`. These methods provide no access to column metadata. That information already will have been disposed of by the time the method returns, and is unavailable to your scripts. However, you can derive column and row counts by examining the result set itself.

PHP

In PHP, metadata for `SELECT` statements is available from PDO after a successful call to `query()`. If you execute a statement using `prepare()` plus `execute()` instead (which can be used for `SELECT` or non-`SELECT` statements), metadata becomes available after `execute()`.

To determine metadata availability, check whether the statement handle `columnCount()` method returns a value greater than zero. If so, the handle's `getColumnMeta()` method returns an associative array containing metadata for a single column. The following table shows the elements of this array. (The format of the `flags` value might differ for other database systems.)

Name	Value
`pdo_type`	Column type (corresponds to a `PDO::PARAM_`*XXX* value)
`native_type`	PHP native type for the column value
`name`	Column name
`len`	Column length
`precision`	Column precision
`flags`	Array of flags describing the column attributes
`table`	Name of table the column is part of

This example code shows how to execute a statement and display result set metadata:

```
$stmt = "SELECT name, birth FROM profile";
print ("Statement: $stmt\n");
$sth = $dbh->prepare ($stmt);
$sth->execute ();
# metadata information becomes available at this point ...
$ncols = $sth->columnCount ();
print ("Number of columns: $ncols\n");
if ($ncols == 0)
  print ("Note: statement has no result set\n");
for ($i = 0; $i < $ncols; $i++)
{
  $col_info = $sth->getColumnMeta ($i);
  $flags = implode (",", array_values ($col_info["flags"]));
  printf ("--- Column %d (%s) ---\n", $i, $col_info["name"]);
  printf ("pdo_type:     %d\n", $col_info["pdo_type"]);
  printf ("native_type:  %s\n", $col_info["native_type"]);
  printf ("len:          %d\n", $col_info["len"]);
  printf ("precision:    %d\n", $col_info["precision"]);
  printf ("flags:        %s\n", $flags);
  printf ("table:        %s\n", $col_info["table"]);
}
```

The program produces this output:

```
Statement: SELECT name, birth FROM profile
Number of columns: 2
--- Column 0 (name) ---
PDO type:     2
native type:  VAR_STRING
len:          20
precision:    0
flags:        not_null
```

```
table:        profile
--- Column 1 (birth) ---
PDO type:     2
native type:  DATE
len:          10
precision:    0
flags:
table:        profile
```

To get a row count from a statement that returns rows, fetch the rows and count them yourself. The `rowCount()` method is not guaranteed to work for result sets.

Python

For statements that produce a result set, Python's DB API makes row and column counts available, as well as a few information items about individual columns.

To get the row count for a result set, access the cursor's `rowcount` attribute. This requires that the cursor be buffered so that it fetches query results immediately; otherwise, you must count the rows as you fetch them. The column count is not available directly, but after calling `fetchone()` or `fetchall()`, you can determine the count as the length of any result set row tuple. It's also possible to determine the column count without fetching any rows by using `cursor.description`. This is a tuple containing one element per column in the result set, so its length tells you how many columns are in the set. (If the statement generates no result set, such as for `UPDATE`, the value of `description` is `None`.) Each element of the `description` tuple is another tuple that represents the metadata for the corresponding column of the result. For Connector/Python, only a few `description` values are meaningful. The following code shows how to access them:

```
stmt = "SELECT name, birth FROM profile"
print("Statement: %s" % stmt)
# buffer cursor so that rowcount has usable value
cursor = conn.cursor(buffered=True)
cursor.execute(stmt)
# metadata information becomes available at this point ...
print("Number of rows: %d" % cursor.rowcount)
if cursor.description is None:  # no result set
  ncols = 0
else:
  ncols = len(cursor.description)
print("Number of columns: %d" % ncols)
if ncols == 0:
  print("Note: statement has no result set")
for i, col_info in enumerate(cursor.description):
  # print name, then other information
  name, type, _, _, _, _, nullable, flags = col_info
  print("--- Column %d (%s) ---" % (i, name))
  print("Type:     %d (%s)" % (type, FieldType.get_info(type)))
  print("Nullable: %d" % (nullable))
```

```
    print("Flags:    %d" % (flags))
cursor.close()
```

The code uses the `FieldType` class, imported as follows:

```
from mysql.connector import FieldType
```

The program produces this output:

```
Statement:  SELECT name, birth FROM profile
Number of rows: 10
Number of columns: 2
--- Column 0 (name) ---
Type:     253 (VAR_STRING)
Nullable: 0
Flags:    4097
--- Column 1 (birth) ---
Type:     10 (DATE)
Nullable: 1
Flags:    128
```

Java

JDBC makes result set metadata available through a `ResultSetMetaData` object, obtained by calling the `getMetaData()` method of your `ResultSet` object. The metadata object provides access to several kinds of information. Its `getColumnCount()` method returns the number of columns in the result set. Other types of metadata, illustrated by the following code, provide information about individual columns and take a column index as their argument. For JDBC, column indexes begin at 1 rather than 0, unlike our other APIs:

```
String stmt = "SELECT name, birth FROM profile";
System.out.println ("Statement: " + stmt);
Statement s = conn.createStatement ();
s.executeQuery (stmt);
ResultSet rs = s.getResultSet ();
ResultSetMetaData md = rs.getMetaData ();
// metadata information becomes available at this point ...
int ncols = md.getColumnCount ();
System.out.println ("Number of columns: " + ncols);
if (ncols == 0)
  System.out.println ("Note: statement has no result set");
for (int i = 1; i <= ncols; i++)  // column index values are 1-based
{
  System.out.println ("--- Column " + i
            + " (" + md.getColumnName (i) + ") ---");
  System.out.println ("getColumnDisplaySize: " + md.getColumnDisplaySize (i));
  System.out.println ("getColumnLabel:       " + md.getColumnLabel (i));
  System.out.println ("getColumnType:        " + md.getColumnType (i));
  System.out.println ("getColumnTypeName:    " + md.getColumnTypeName (i));
  System.out.println ("getPrecision:         " + md.getPrecision (i));
  System.out.println ("getScale:             " + md.getScale (i));
```

```
    System.out.println ("getTableName:         " + md.getTableName (i));
    System.out.println ("isAutoIncrement:      " + md.isAutoIncrement (i));
    System.out.println ("isNullable:           " + md.isNullable (i));
    System.out.println ("isCaseSensitive:      " + md.isCaseSensitive (i));
    System.out.println ("isSigned:             " + md.isSigned (i));
  }
  rs.close ();
  s.close ();
```

The program produces this output:

```
Statement: SELECT name, birth FROM profile
Number of columns: 2
--- Column 1 (name) ---
getColumnDisplaySize: 20
getColumnLabel:       name
getColumnType:        12
getColumnTypeName:    VARCHAR
getPrecision:         20
getScale:             0
getTableName:         profile
isAutoIncrement:      false
isNullable:           0
isCaseSensitive:      false
isSigned:             false
--- Column 2 (birth) ---
getColumnDisplaySize: 10
getColumnLabel:       birth
getColumnType:        91
getColumnTypeName:    DATE
getPrecision:         10
getScale:             0
getTableName:         profile
isAutoIncrement:      false
isNullable:           1
isCaseSensitive:      false
isSigned:             false
```

The row count of the result set is not available directly; you must fetch the rows and count them.

JDBC has several other result set metadata calls, but many of them provide no useful information for MySQL. To try them, get a JDBC reference to see what the calls are and modify the program to see what, if anything, they return.

10.3. Determining Whether a Statement Produced a Result Set

Problem

You just executed an SQL statement, but you're not sure whether it produced a result set.

Solution

Check the column count in the metadata. There is no result set if the count is zero.

Discussion

If you write an application that accepts statement strings from an external source such as a file or a user entering text at the keyboard, you may not necessarily know whether it's a statement such as `SELECT` that produces a result set or a statement such as `UPDATE` that does not. That's an important distinction because you process statements that produce a result set differently from those that do not. Assuming that no error occurred, one way to tell the difference is to check the metadata value that indicates the column count after executing the statement (as shown in Recipe 10.2). A column count of zero indicates that the statement was an `INSERT`, `UPDATE`, or some other statement that returns no result set. A nonzero value indicates the presence of a result set, and you can go ahead and fetch the rows. This technique distinguishes `SELECT` from non-`SELECT` statements, even for `SELECT` statements that return an empty result set. (An empty result is different from no result. The former returns no rows, but the column count is still correct; the latter has no columns at all.)

Some APIs provide ways to distinguish statement types other than checking the column count:

- In Python, the value of `cursor.description` is `None` for statements that produce no result set.
- In JDBC, you can issue arbitrary statements using the `execute()` method, which returns true or false to indicate whether there is a result set.

10.4. Using Metadata to Format Query Output

Problem

You want to display a result set, nicely formatted.

Solution

Let the result set metadata help you. It provides important information about the structure and content of the results.

Discussion

Metadata is valuable for formatting query results because it tells you several important things about the columns, such as the names and display widths. For example, you can write a general-purpose function that displays a result set in tabular format, even without knowing what the query was. The following Java code shows one way to do this. It takes a result set object and uses it to get the metadata for the result. Then it uses both objects in tandem to retrieve and format the values in the result. The output is similar to that produced by *mysql*: a row of column headers followed by the rows of the result, with columns nicely boxed and lined up vertically. Here's a sample of function output, given the result set generated by the query `SELECT id, name, birth FROM profile`:

```
+----------+-------------------+----------+
|id        |name               |birth     |
+----------+-------------------+----------+
|1         |Sybil              |1970-04-13|
|2         |Nancy              |1969-09-30|
|3         |Ralph              |1973-11-02|
|4         |Lothair            |1963-07-04|
|5         |Henry              |1965-02-14|
|6         |Aaron              |1968-09-17|
|7         |Joanna             |1952-08-20|
|8         |Stephen            |1960-05-01|
|9         |Amabel             |NULL      |
+----------+-------------------+----------+
Number of rows selected: 9
```

The primary problem an application like this must solve is to determine the proper display width of each column. The `getColumnDisplaySize()` method returns the column width, but we must also factor in other pieces of information:

- The column name might be longer than the column width.
- We'll print the word "NULL" for `NULL` values, so if the column can contain `NULL` values, the display width must be at least four.

The following Java function, `displayResultSet()`, formats a result set, taking those factors into account. It also counts rows as it fetches them to determine the row count, because JDBC doesn't provide that value in the metadata:

```
public static void displayResultSet (ResultSet rs) throws SQLException
{
  ResultSetMetaData md = rs.getMetaData ();
  int ncols = md.getColumnCount ();
```

```
int nrows = 0;
int[] width = new int[ncols + 1];   // array to store column widths
StringBuffer b = new StringBuffer (); // buffer to hold bar line

// calculate column widths
for (int i = 1; i <= ncols; i++)
{
  // some drivers return -1 for getColumnDisplaySize();
  // if so, we'll override that with the column name length
  width[i] = md.getColumnDisplaySize (i);
  if (width[i] < md.getColumnName (i).length ())
    width[i] = md.getColumnName (i).length ();
  // isNullable() returns 1/0, not true/false
  if (width[i] < 4 && md.isNullable (i) != 0)
    width[i] = 4;
}

// construct +---+---...+ line
b.append ("+");
for (int i = 1; i <= ncols; i++)
{
  for (int j = 0; j < width[i]; j++)
    b.append ("-");
  b.append ("+");
}

// print bar line, column headers, bar line
System.out.println (b.toString ());
System.out.print ("|");
for (int i = 1; i <= ncols; i++)
{
  System.out.print (md.getColumnName (i));
  for (int j = md.getColumnName (i).length (); j < width[i]; j++)
    System.out.print (" ");
  System.out.print ("|");
}
System.out.println ();
System.out.println (b.toString ());

// print contents of result set
while (rs.next ())
{
  ++nrows;
  System.out.print ("|");
  for (int i = 1; i <= ncols; i++)
  {
    String s = rs.getString (i);
    if (rs.wasNull ())
      s = "NULL";
    System.out.print (s);
    for (int j = s.length (); j < width[i]; j++)
      System.out.print (" ");
```

```
        System.out.print ("|");
      }
      System.out.println ();
    }
    // print bar line, and row count
    System.out.println (b.toString ());
    System.out.println ("Number of rows selected: " + nrows);
  }
```

To be more elaborate, test whether a column contains numeric values and format it as right-justified if so. In Perl DBI scripts, this is easy to check because you can access the `mysql_is_num` metadata attribute. For other APIs, it is not so easy unless there is some equivalent "column is numeric" metadata value available. If not, you must check whether the data-type indicator is one of the several possible numeric types.

The `displayResultSet()` function prints columns using the width of the column as specified in the table definition, not the maximum width of the values actually present in the result set. The latter value is often smaller. You can see this in the sample output that precedes the listing for `displayResultSet()`. The `id` and `name` columns are 10 and 20 characters wide, even though the widest values are only two and seven characters long, respectively. In Perl and Ruby, you can get the maximum width of the values present in the result set. To determine these widths in JDBC, you must iterate through the result set and check the column value lengths yourself. This requires a JDBC 2.0 driver that provides scrollable result sets. If you have such a driver (Connector/J is one), the column-width calculation code in the `displayResultSet()` function can be modified as follows:

```
// calculate column widths
for (int i = 1; i <= ncols; i++)
{
  width[i] = md.getColumnName (i).length ();
  // isNullable() returns 1/0, not true/false
  if (width[i] < 4 && md.isNullable (i) != 0)
    width[i] = 4;
}
// scroll through result set and adjust display widths as necessary
while (rs.next ())
{
  for (int i = 1; i <= ncols; i++)
  {
    byte[] bytes = rs.getBytes (i);
    if (!rs.wasNull ())
    {
      int len = bytes.length;
      if (width[i] < len)
        width[i] = len;
    }
  }
}
rs.beforeFirst ();  // rewind result set before displaying it
```

With that change, the result is a more compact query result display:

```
+--+-------+----------+
|id|name   |birth     |
+--+-------+----------+
|1 |Sybil  |1970-04-13|
|2 |Nancy  |1969-09-30|
|3 |Ralph  |1973-11-02|
|4 |Lothair|1963-07-04|
|5 |Henry  |1965-02-14|
|6 |Aaron  |1968-09-17|
|7 |Joanna |1952-08-20|
|8 |Stephen|1960-05-01|
|9 |Amabel |NULL      |
+--+-------+----------+
Number of rows selected: 9
```

Before writing your own function, check whether your API already provides one. For example, the Ruby DBI::Utils::TableFormatter module has an `ascii` method that produces a formatted display much like that just described. Use it like this:

```
dbh.execute(stmt) do |sth|
  DBI::Utils::TableFormatter.ascii(sth.column_names, sth.fetch_all)
end
```

10.5. Listing or Checking Existence of Databases or Tables

Problem

You want to list the databases hosted by the MySQL server or the tables in a database. Or you want to check whether a particular database or table exists.

Solution

Use `INFORMATION_SCHEMA` to get this information. The `SCHEMATA` table contains a row for each database, and the `TABLES` table contains a row for each table in each database.

Discussion

To retrieve the list of databases hosted by the server, use this statement:

```
SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA;
```

To sort the result, add an `ORDER BY SCHEMA_NAME` clause.

To check whether a specific database exists, use a `WHERE` clause with a condition that names the database. If you get a row back, the database exists. The following Ruby method shows how to perform an existence test for a database:

```
def database_exists(dbh, db_name)
  return !dbh.select_one("SELECT SCHEMA_NAME
                          FROM INFORMATION_SCHEMA.SCHEMATA
                          WHERE SCHEMA_NAME = ?", db_name).nil?
end
```

To obtain the list of tables in a database, name the database in the `WHERE` clause of a statement that selects from the `TABLES` table:

```
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'cookbook';
```

To sort the result, add an `ORDER BY TABLE_NAME` clause.

To obtain a list of tables in the default database, use this statement instead:

```
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = DATABASE();
```

If no database has been selected, `DATABASE()` returns `NULL` and no rows match, which is the correct result.

To check whether a specific table exists, use a `WHERE` clause with a condition that names the table. Here's a Ruby method that performs an existence test for a table in a given database:

```
def table_exists(dbh, db_name, tbl_name)
  return !dbh.select_one("SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
                          WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?",
                         db_name, tbl_name).nil?
end
```

Some APIs provide a database-independent way to get database or table lists. In Perl DBI, the database handle `tables()` method returns a list of tables in the default database:

```
@tables = $dbh->tables ();
```

The Ruby method is similar:

```
tables = dbh.tables
```

For Java, there are JDBC methods designed to return lists of databases or tables. For each method, invoke your connection object's `getMetaData()` method and use the resulting `DatabaseMetaData` object to retrieve the information you want. Here's how to produce a list of databases:

```
// get list of databases
DatabaseMetaData md = conn.getMetaData ();
ResultSet rs = md.getCatalogs ();
while (rs.next ())
  System.out.println (rs.getString (1));  // column 1 = database name
rs.close ();
```

To list the tables in a database, do this:

```
// get list of tables in database named by dbName; if
// dbName is the empty string, the default database is used
DatabaseMetaData md = conn.getMetaData ();
ResultSet rs = md.getTables (dbName, "", "%", null);
while (rs.next ())
  System.out.println (rs.getString (3));  // column 3 = table name
rs.close ();
```

10.6. Accessing Table Column Definitions

Problem

You want to find out what columns a table has and how they are defined.

Solution

There are several ways to do this. You can obtain column definitions from `INFORMATION_SCHEMA`, from `SHOW` statements, or from *mysqldump*.

Discussion

Information about the structure of tables enables you to answer questions such as "What columns does a table contain and what are their types?" or "What are the legal values for an `ENUM` or `SET` column?" Here are some applications for that kind of information:

Displaying column lists
: A simple use of table information is presenting a list of the table's columns. This is common in web-based or GUI applications that enable users to construct statements interactively by selecting a table column from a list and entering a value against which to compare column values.

Interactive record editing
: Knowledge of a table's structure can be very useful for interactive record-editing applications. Suppose that an application retrieves a record from the database, displays a form containing the record's content so a user can edit it, and then updates the record in the database after the user modifies the form and submits it. You can use table structure information for validating column values. If a column is an `ENUM`, you can find out the valid enumeration values and check the value submitted by the user against them to determine whether it's legal. If the column is an integer type, check the submitted value to make sure that it consists entirely of digits, possibly preceded by a `+` or `–` sign character. If the column contains dates, look for a legal date format.

  But what if the user leaves a field empty? If the field corresponds to, say, a `CHAR` column in the table, do you set the column value to `NULL` or to the empty string? This too is a question that can be answered by checking the table's structure.

Determine whether the column can contain NULL values. If it can, set the column to NULL; otherwise, set it to the empty string.

Mapping column definitions onto web page elements
Some data types such as ENUM and SET correspond naturally to elements of web forms:

- An ENUM has a fixed set of values from which you choose a single value. This is analogous to a group of radio buttons, a pop-up menu, or a single-pick scrolling list.
- A SET column is similar, except that you can select multiple values; this corresponds to a group of checkboxes or a multiple-pick scrolling list.

By using table metadata to access definitions for these types of columns, you can easily determine a column's legal values and map them onto an appropriate form element. This enables you to present users with a list of applicable values from which they can make a selection easily with no typing. Recipe 10.7 discusses how to get definitions for these types of columns. The methods developed there are used in Chapter 20, which discusses form generation in more detail.

MySQL provides several ways to find out about a table's structure:

- Retrieve the information from INFORMATION_SCHEMA. The COLUMNS table contains the column definitions.
- Use a SHOW COLUMNS statement.
- Use the SHOW CREATE TABLE statement or the *mysqldump* command-line program to obtain a CREATE TABLE statement that displays the table's structure.

The following discussion shows how to ask MySQL for table information using each method. To try the examples, create an item table that lists item IDs, names, and colors in which each item is available:

```
CREATE TABLE item
(
  id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
  name    CHAR(20),
  colors  ENUM('chartreuse','mauve','lime green','puce') DEFAULT 'puce',
  PRIMARY KEY (id)
);
```

Using INFORMATION_SCHEMA to get table structure

To obtain information about a single column in a table by checking INFORMATION_SCHEMA, use a statement of the following form:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'item'
```

```
    -> AND COLUMN_NAME = 'colors'\G
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: cookbook
              TABLE_NAME: item
             COLUMN_NAME: colors
        ORDINAL_POSITION: 3
          COLUMN_DEFAULT: puce
             IS_NULLABLE: YES
               DATA_TYPE: enum
CHARACTER_MAXIMUM_LENGTH: 10
  CHARACTER_OCTET_LENGTH: 10
       NUMERIC_PRECISION: NULL
           NUMERIC_SCALE: NULL
      DATETIME_PRECISION: NULL
      CHARACTER_SET_NAME: latin1
          COLLATION_NAME: latin1_swedish_ci
             COLUMN_TYPE: enum('chartreuse','mauve','lime green','puce')
              COLUMN_KEY:
                   EXTRA:
              PRIVILEGES: select,insert,update,references
          COLUMN_COMMENT:
```

To obtain information about all columns, omit the `COLUMN_NAME` condition from the `WHERE` clause.

To retrieve only certain types of information, replace `SELECT *` with the columns of interest:

```
mysql> SELECT COLUMN_NAME, DATA_TYPE, IS_NULLABLE
    -> FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'item';
+-------------+-----------+-------------+
| COLUMN_NAME | DATA_TYPE | IS_NULLABLE |
+-------------+-----------+-------------+
| id          | int       | NO          |
| name        | char      | YES         |
| colors      | enum      | YES         |
+-------------+-----------+-------------+
```

Here are some `COLUMNS` table columns likely to be of most use:

- `COLUMN_NAME`: The column name.
- `ORDINAL_POSITION`: The position of the column within the table definition.
- `COLUMN_DEFAULT`: The column's default value.
- `IS_NULLABLE`: `YES` or `NO` to indicate whether the column can contain `NULL` values.
- `DATA_TYPE`, `COLUMN_TYPE`: Data type information. `DATA_TYPE` is the data-type keyword and `COLUMN_TYPE` contains additional information such as type attributes.

- CHARACTER_SET_NAME, COLLATION_NAME: The character set and collation for string columns. They are NULL for nonstring columns.
- COLUMN_KEY: Information about whether the column is indexed.

INFORMATION_SCHEMA content is easy to use from within programs. Here's a PHP function that illustrates this process. It takes database and table name arguments, selects from INFORMATION_SCHEMA to obtain a list of the table's column names, and returns the names as an array. The ORDER BY ORDINAL_POSITION clause ensures that names in the array are returned in table-definition order:

```
function get_column_names ($dbh, $db_name, $tbl_name)
{
  $stmt = "SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS
           WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?
           ORDER BY ORDINAL_POSITION";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ($db_name, $tbl_name));
  return ($sth->fetchAll (PDO::FETCH_COLUMN, 0));
}
```

get_column_names() returns an array containing only column names. If you require additional column information, it's possible to write a more general get_column_info() routine that returns an array of column information structures. For implementations of both routines in PHP as well as other languages, check the library files in the *lib* directory of the recipes distribution.

Using SHOW COLUMNS to get table structure

The SHOW COLUMNS statement produces one row of output for each column in the table, with each row providing various pieces of information about the corresponding column. The following example demonstrates SHOW COLUMNS output for the item table colors column:

```
mysql> SHOW COLUMNS FROM item LIKE 'colors'\G
*************************** 1. row ***************************
  Field: colors
   Type: enum('chartreuse','mauve','lime green','puce')
   Null: YES
    Key:
Default: puce
  Extra:
```

SHOW COLUMNS displays information for all columns having a name that matches the LIKE pattern. To obtain information about all columns, omit the LIKE clause.

The values displayed by SHOW COLUMNS correspond to these columns of the INFORMATION_SCHEMA COLUMNS table: COLUMN_NAME, COLUMN_TYPE, COLUMN_KEY, IS_NULLABLE, COLUMN_DEFAULT, EXTRA.

SHOW FULL COLUMNS displays additional Collation, Privileges, and Comment fields for each column. These correspond to the COLUMNS table COLLATION_NAME, PRIVILEGES, and COLUMN_COMMENT columns.

SHOW interprets the pattern the same way as for the LIKE operator in the WHERE clause of a SELECT statement. (For information about pattern matching, see Recipe 5.8.) If you specify a literal column name, the string matches only that name and SHOW COLUMNS displays information only for that column. However, a trap awaits the unwary here. If your column name contains SQL pattern characters (% or _) that you want to match literally, you must escape them with a backslash in the pattern string to avoid matching other names as well.

The need to escape % and _ characters to match a LIKE pattern literally also applies to other SHOW statements that permit a name pattern in the LIKE clause, such as SHOW TABLES and SHOW DATABASES.

Within a program, you can use your API language's pattern-matching capabilities to escape SQL pattern characters before putting the column name into a SHOW statement. In Perl, Ruby, and PHP, use the following expressions.

Perl:

```
$name =~ s/([%_])/\\$1/g;
```

Ruby:

```
name.gsub!(/([%_])/, '\\\\1')
```

PHP:

```
$name = preg_replace ('/([%_])/', '\\\\$1', $name);
```

For Python, import the re module, and use its sub() method:

```
name = re.sub(r'([%_])', r'\\1', name)
```

For Java, use methods from the java.util.regex package:

```
import java.util.regex.*;

Pattern p = Pattern.compile("([_%])");
Matcher m = p.matcher(name);
name = m.replaceAll ("\\\\$1");
```

If these expressions appear to have too many backslashes, remember that the API language processor itself interprets backslashes and strips off a level before performing the pattern match. To get a literal backslash into the result, it must be doubled in the pattern. Another level on top of that is needed if the pattern processor strips a set.

Using CREATE TABLE to get table structure

Another way to obtain table structure information from MySQL is from the `CREATE TABLE` statement that defines the table. To get this information, use the `SHOW CREATE TABLE` statement:

```
mysql> SHOW CREATE TABLE item\G
*************************** 1. row ***************************
       Table: item
Create Table: CREATE TABLE `item` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` char(20) DEFAULT NULL,
  `colors` enum('chartreuse','mauve','lime green','puce') DEFAULT 'puce',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

From the command line, the same `CREATE TABLE` information is available from *mysqldump* if you use the `--no-data` option, which tells *mysqldump* to dump only the structure of the table and not its data.

`CREATE TABLE` format is highly informative and easy to read because it shows column information in a format similar to the one you used to create the table in the first place. It also shows the index structure clearly, whereas the other methods do not. However, you'll probably find this method of checking table structure more useful interactively than within programs. The information isn't provided in regular row-and-column format, so it's more difficult to parse. Also, the format is subject to change whenever the `CREATE TABLE` statement is enhanced, which happens from time to time as MySQL's capabilities are extended.

10.7. Getting ENUM and SET Column Information

Problem

You want to know the members of an `ENUM` or `SET` column.

Solution

This problem is a subset of getting table structure metadata. Obtain the column definition from the table metadata, then extract the member list from the definition.

Discussion

It's often useful to know the list of legal values for an `ENUM` or `SET` column. Suppose that you want to present a web form containing a pop-up menu that has options corresponding to each legal value of an `ENUM` column, such as the sizes in which a garment can be ordered, or the available shipping methods for delivering a package. You could hardwire the choices into the script that generates the form, but if you alter the column

later (for example, to add a new enumeration value), you introduce a discrepancy between the column and the script that uses it. If instead you look up the legal values using the table metadata, the script can always produce a pop-up that contains the proper set of values. A similar approach applies to `SET` columns.

To determine the permitted values for an `ENUM` or `SET` column, get its definition using one of the techniques described in Recipe 10.6. For example, if you select from the `INFORMATION_SCHEMA COLUMNS` table, the `COLUMN_TYPE` value for the `colors` column of the `item` table looks like this:

```
enum('chartreuse','mauve','lime green','puce')
```

`SET` columns are similar, except that they say `set` rather than `enum`. For either data type, extract the permitted values by stripping the initial word and the parentheses, splitting at the commas, and removing the enclosing quotes from the individual values. Let's write a `get_enumorset_info()` routine to break out these values from the data-type definition. While we're at it, we can have the routine return the column's type, its default value, and whether values can be `NULL`. Then the routine can be used by scripts that may need more than just the list of values. Here is a version in Ruby. Its arguments are a database handle, a database name, a table name, and a column name. It returns a hash with entries corresponding to the various aspects of the column definition (or `nil` if the column does not exist):

```
def get_enumorset_info(dbh, db_name, tbl_name, col_name)
  row = dbh.select_one(
          "SELECT COLUMN_NAME, COLUMN_TYPE, IS_NULLABLE, COLUMN_DEFAULT
           FROM INFORMATION_SCHEMA.COLUMNS
           WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ? AND COLUMN_NAME = ?",
          db_name, tbl_name, col_name)
  return nil if row.nil?  # no such column
  info = {}
  info["name"] = row[0]
  return nil unless row[1] =~ /^(ENUM|SET)\((.*)\)$/i # not ENUM or SET
  info["type"] = $1
  # split value list on commas, trim quotes from end of each word
  info["values"] = $2.split(",").collect { |val| val.sub(/^'(.*)'$/, "\\1") }
  # determine whether column can contain NULL values
  info["nullable"] = (row[2].upcase == "YES")
  # get default value (nil represents NULL)
  info["default"] = row[3]
  return info
end
```

The routine uses case-insensitive matching when checking the data type and nullable attributes. This guards against future lettercase changes in metadata results.

The following example shows how to access and display each element of the hash returned by `get_enumorset_info()`:

```
info = get_enumorset_info(dbh, db_name, tbl_name, col_name)
puts "Information for #{db_name}.#{tbl_name}.#{col_name}:"
if info.nil?
  puts "No information available (not an ENUM or SET column?)"
else
  puts "Name: " + info["name"]
  puts "Type: " + info["type"]
  puts "Legal values: " + info["values"].join(",")
  puts "Nullable: " + (info["nullable"] ? "yes" : "no")
  puts "Default value: " + (info["default"].nil? ? "NULL" : info["default"])
end
```

That code produces the following output for the `item` table `colors` column:

```
Information for cookbook.item.colors:
Name: colors
Type: enum
Legal values: chartreuse,mauve,lime green,puce
Nullable: yes
Default value: puce
```

Equivalent routines for other APIs are similar. You can find implementations in the *lib* directory of the `recipes` distribution. Such routines are useful for validation of input values (see Recipe 12.8), and are especially handy for generating list elements in web forms (see Recipes 20.2 and 20.3).

10.8. Getting Server Metadata

Problem

You want the MySQL server to tell you about itself.

Solution

Several SQL functions and `SHOW` statements return information about the server.

Discussion

MySQL has several SQL functions and statements that provide you with information about the server itself and about your current client session. The following table shows a few that you may find useful. Both `SHOW` statements permit a `GLOBAL` or `SESSION` keyword to select global server values or values specific to your session, and a `LIKE '`*`pattern`*`'` clause for limiting the results to variable names matching the pattern:

Statement	Information produced by statement
`SELECT VERSION()`	Server version string
`SELECT DATABASE()`	Default database name (`NULL` if none)

Statement	Information produced by statement
`SELECT USER()`	Current user as given by client when connecting
`SELECT CURRENT_USER()`	User used for checking client privileges
`SHOW [GLOBAL|SESSION] STATUS`	Server global or session status indicators
`SHOW [GLOBAL|SESSION] VARIABLES`	Server global or status configuration variables

To obtain the information provided by any statement in the table, execute it and process its result set. For example, `SELECT DATABASE()` returns the name of the default database or `NULL` if no database has been selected. The following Ruby code uses the statement to present a status display containing information about the current session:

```
db = dbh.select_one("SELECT DATABASE()")[0]
puts "Default database: " + (db.nil? ? "(no database selected)" : db)
```

A given API might provide alternatives to executing SQL statements to access these types of information. For example, JDBC has several database-independent methods for obtaining server metadata. Use your connection object to obtain the database metadata, then invoke the appropriate methods to get the information in which you're interested. Consult a JDBC reference for a complete list, but here are a few representative examples:

```
DatabaseMetaData md = conn.getMetaData ();
// can also get this with SELECT VERSION()
System.out.println ("Product version: " + md.getDatabaseProductVersion ());
// this is similar to SELECT USER() but doesn't include the hostname
System.out.println ("Username: " + md.getUserName ());
```

See Also

For more discussion about the use of `SHOW` (and `INFORMATION_SCHEMA`) in the context of server monitoring, see Recipe 22.6.

10.9. Writing Applications That Adapt to the MySQL Server Version

Problem

You want to use a given feature that is available only as of a particular version of MySQL.

Solution

Ask the server for its version number. If the server is too old to support a given feature, maybe you can fall back to a workaround, if one exists.

Discussion

Over the course of MySQL development, new versions add features. If you're writing an application that requires certain features, check the server version to determine whether they are present; if not, you must perform some sort of workaround (assuming there is one).

To get the server version, invoke the `VERSION()` function. The result is a string that looks something like `5.5.25a` or `5.7.4-m14`. In other words, it returns a string consisting of major, minor, and "teeny" version numbers, possibly some nondigits at the end of the "teeny" version, and possibly some suffix. The version string can be used as is for presentation purposes, but for comparisons, it's simpler to work with a number—in particular, a five-digit number in *`Mmmtt`* format, in which *`M`*, *`mm`*, *`tt`* are the major, minor, and teeny version numbers. Perform the conversion by splitting the string at the periods, stripping from the third piece the suffix that begins with the first nonnumeric character, and joining the pieces. For example, `5.5.25a` becomes `50525`, and `5.7.4-m14` becomes `50704`.

Here's a Perl DBI function that takes a database-handle argument and returns a two-element list that contains both the string and numeric forms of the server version. The code assumes that the minor and teeny version parts are less than 100 and thus no more than two digits each. That should be a valid assumption because the source code for MySQL itself uses the same format:

```
sub get_server_version
{
my $dbh = shift;
my ($ver_str, $ver_num);
my ($major, $minor, $teeny);

  # fetch result into scalar string
  $ver_str = $dbh->selectrow_array ("SELECT VERSION()");
  return undef unless defined ($ver_str);
  ($major, $minor, $teeny) = split (/\./, $ver_str);
  $teeny =~ s/\D.*$//; # strip nonnumeric suffix if present
  $ver_num = $major*10000 + $minor*100 + $teeny;
  return ($ver_str, $ver_num);
}
```

To get both forms of the version information at once, call the function like this:

```
my ($ver_str, $ver_num) = get_server_version ($dbh);
```

To get just one of the values, call it as follows:

```
my $ver_str = (get_server_version ($dbh))[0]; # string form
my $ver_num = (get_server_version ($dbh))[1]; # numeric form
```

The following examples demonstrate how to use the numeric version value to check whether the server supports certain features:

```
my $ver_num = (get_server_version ($dbh))[1];
printf "Event scheduler:    %s\n", ($ver_num >= 50106 ? "yes" : "no");
printf "4-byte Unicode:     %s\n", ($ver_num >= 50503 ? "yes" : "no");
printf "Fractional seconds: %s\n", ($ver_num >= 50604 ? "yes" : "no");
printf "SHA-256 passwords:  %s\n", ($ver_num >= 50606 ? "yes" : "no");
printf "ALTER USER:         %s\n", ($ver_num >= 50607 ? "yes" : "no");
printf "INSERT DELAYED:     %s\n", ($ver_num >= 50700 ? "no" : "yes");
```

The `recipes` distribution *metadata* directory contains `get_server_version()` implementations in other API languages, and the *routines* directory contains a `server_version()` stored function for use in SQL statements. The latter function returns only the numeric value because `VERSION()` already produces the string value. The following example shows how to use it to implement a stored procedure that expires an account password if the server is recent enough to support the `ALTER USER` statement (MySQL 5.6.7 or later):

```
CREATE PROCEDURE expire_password(user TEXT, host TEXT)
BEGIN
  DECLARE account TEXT;
  SET account = CONCAT(QUOTE(user),'@',QUOTE(host));
  IF server_version() >= 50607 AND user <> '' THEN
    CALL exec_stmt(CONCAT('ALTER USER ',account,' PASSWORD EXPIRE'));
  END IF;
END;
```

`expire_password()` requires the `exec_stmt()` helper routine (see Recipe 9.9). Both are available in the *routines* directory. For more information about password expiration, see Recipe 23.5.

CHAPTER 11
Importing and Exporting Data

11.0. Introduction

Suppose that a file named *somedata.csv* contains 12 data columns in comma-separated values (CSV) format. From this file you want to extract only columns 2, 11, 5, and 9, and use them to create database rows in a MySQL table that contains `name`, `birth`, `height`, and `weight` columns. You must make sure that the height and weight are positive integers, and convert the birth dates from *MM/DD/YY* format to *CCYY-MM-DD* format. How can you do this?

In one sense, that problem is very specialized. But it's not at all atypical because data transfer problems with specific requirements occur frequently when you transfer data into MySQL. Datafiles are not always nicely formatted and ready to load into MySQL with no preparation. As a result, it's often necessary to preprocess information to put it into a format acceptable for MySQL. The reverse also is true; data exported from MySQL may need massaging to be useful for other programs.

Although some data preparation operations are so difficult that they require a great deal of hand checking and reformatting, in most cases you can do at least part of the job automatically. Virtually all such problems involve at least some elements of a common set of conversion issues. This chapter and the next discuss what these issues are, how to deal with them by taking advantage of the existing tools at your disposal, and how to write your own tools when necessary. The idea is not to cover all possible situations (an impossible task), but to show representative techniques and utilities. Use them as is or adapt them for problems that they don't handle. (There are commercial data-handling tools, but my purpose here is to enable you to do things yourself.) With respect to the problem posed at the beginning of this Introduction, see Recipe 12.15 for the solution we arrive at.

The discussion begins with native MySQL facilities for importing data (the `LOAD DATA` statement and the *mysqlimport* command-line program), and for exporting data (the

`SELECT ... INTO OUTFILE` statement). For situations where the native facilities do not suffice, we move on to cover techniques for using external supporting utilities (such as *sed* and *tr*) and for writing your own. There are two broad sets of issues to consider:

- How to manipulate the *structure* of datafiles. When a file is in a format not suitable for import, you must convert it to a different format. This may involve issues such as changing the column delimiters or line-ending sequences, or removing or rearranging columns in the file. This chapter covers such techniques.
- How to manipulate the *content* of datafiles. If you don't know whether the values contained in a file are legal, you may want to preprocess it to check or reformat them. Numeric values may need verification as lying within a specific range, dates may need conversion to or from ISO format, and so forth. Chapter 12 covers those techniques.

Source code for program fragments and scripts discussed in this chapter is located in the *transfer* directory of the `recipes` distribution.

General Import and Export Issues

Incompatible datafile formats and differing rules for interpreting various kinds of values cause headaches when transferring data between programs. Nevertheless, certain issues recur frequently. Be aware of them and you can identify more easily what must be done to solve particular import or export problems.

In its most basic form, an input stream is just a set of bytes with no particular meaning. Successful import into MySQL requires recognizing which bytes represent structural information and which represent the data values framed by that structure. Because such recognition is key to decomposing the input into appropriate units, the most fundamental import issues are these:

- What is the record separator? Knowing this enables you to partition the input stream into records.
- What is the field delimiter? Knowing this enables you to partition each record into field values. Identifying the data values also might include stripping quotes from around the values or recognizing escape sequences within them.

The ability to break the input into records and fields is important for extracting the data values from it. If the values are still not in a form that can be used directly, you may need to consider other issues:

- Do the order and number of columns match the structure of the database table? Mismatches require rearranging or skipping columns.

- How should `NULL` or empty values be handled? Are they permitted? Can `NULL` values even be detected? (Some systems export `NULL` values as empty strings, making it impossible to distinguish them.)
- Do data values require validation or reformatting? If the values are in a format that matches MySQL's expectations, no further processing is necessary. Otherwise, they must be checked and possibly rewritten.

For export from MySQL, the issues are somewhat the reverse. You can assume that values stored in the database are valid, but it's necessary to add column and record delimiters to form an output stream that has a structure other programs can recognize, and values may require reformatting for use by other programs.

File Formats

Datafiles come in many formats, two of which appear frequently in this chapter:

Tab-delimited or tab-separated values (TSV) format
: This is one of the simplest file structures; lines contain values separated by tab characters. A short tab-delimited file might look like this, where the whitespace between column values represents single tab characters:

```
a       b       c
a,b,c   d e     f
```

Comma-separated values (CSV) format
: Files written in CSV format vary somewhat; there is apparently no formal standard describing the format. However, the general idea is that lines consist of values separated by commas, and values containing internal commas are enclosed within quotes to prevent the commas from being interpreted as value delimiters. It's also common for values containing spaces to be quoted as well. In this example, each line contains three values:

```
a,b,c
"a,b,c","d e",f
```

: It's trickier to process CSV files than tab-delimited files because characters like quotes and commas have a dual meaning: they may represent file structure or be included in the content of data values.

Another important datafile characteristic is the line-ending sequence. The most common sequences are carriage return, linefeed, and carriage return/linefeed pair, sometimes referred to here by the abbreviations CR, LF, and CRLF.

Datafiles often begin with a row of column labels. For some import operations, the row of labels must be discarded to avoid having it be loaded into your table as data. In other cases, the labels are quite useful:

- For import into existing tables, the labels help you match datafile columns with the table columns if they are not necessarily in the same order.
- The labels can be used for column names when creating a new table automatically or semiautomatically from a datafile. For example, Recipe 11.11 discusses a utility that examines a datafile and guesses the `CREATE TABLE` statement to use to create a table from the file. If a label row is present, the utility uses the labels for column names.

Tab-Delimited, Linefeed-Terminated Format

Although datafiles may be written in many formats, it's unwieldy to include machinery for reading multiple formats within each file-processing utility you write. For that reason, many of the utilities described in this chapter assume for simplicity that their input is in tab-delimited, linefeed-terminated format. (This is also the default format for MySQL's `LOAD DATA` statement.) By making this assumption, it becomes easier to write programs that read files.

On the other hand, *something* has to be able to read data in other formats. To handle that problem, we'll develop a *cvt_file.pl* script that can read several types of files (see Recipe 11.6). The script is based on the Perl Text::CSV_XS module, which despite its name is useful for much more than just CSV data. *cvt_file.pl* can convert between many file types, making it possible for other programs that require tab-delimited lines to be used with files not originally written in that format. In other words, you can use *cvt_file.pl* to convert a file to tab-delimited, linefeed-terminated format, and then any program that expects that format can process the file.

Notes on Invoking Shell Commands

This chapter shows a number of programs that you invoke from the command line using a shell like *bash* or *tcsh* under Unix or *cmd.exe* ("the command prompt") under Windows. Many of the example commands for these programs use quotes around option values, and sometimes an option value is itself a quote character. Quoting conventions vary from one shell to another, but the following rules seem to work with most of them (including *cmd.exe* under Windows):

- For an argument that contains spaces, enclose it within double quotes to prevent the shell from interpreting it as multiple separate arguments. The shell strips the quotes and passes the argument to the command intact.
- To include a double-quote character in the argument itself, precede it with a backslash.

Some shell commands in this chapter are so long that they're shown as you would enter them using several lines, with a backslash character as the line-continuation character:

```
% prog_name \
    argument1 \
    argument2 ...
```

That works for Unix. On Windows, the continuation character is ^ (or ` for PowerShell). Alternatively, on any platform, enter the entire command on one line:

```
C:\> prog_name argument1 argument2 ...
```

11.1. Importing Data with LOAD DATA and mysqlimport

Problem

You want to load a datafile into a table using MySQL's built-in import capabilities.

Solution

Use the `LOAD DATA` statement or the *mysqlimport* command-line program.

Discussion

MySQL provides a `LOAD DATA` statement that acts as a bulk data loader. Here's an example statement that reads a file *mytbl.txt* from your current directory and loads it into the table `mytbl` in the default database:

```
mysql> LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl;
```

At some MySQL installations, the `LOCAL` loading capability may have been disabled for security reasons. If that is true at your site, omit `LOCAL` from the statement and specify the full pathname to the file, which must be readable by the server. Local versus nonlocal data loading is discussed shortly.

The MySQL utility program *mysqlimport* acts as a wrapper around `LOAD DATA` so that you can load input files directly from the command line. The *mysqlimport* command that is equivalent to the preceding `LOAD DATA` statement looks like this, assuming that `mytbl` is in the `cookbook` database:

```
% mysqlimport --local cookbook mytbl.txt
```

For *mysqlimport*, as with other MySQL programs, you may need to specify connection parameter options such as `--user` or `--host` (see Recipe 1.4).

The following list describes `LOAD DATA`'s general characteristics and capabilities; *mysqlimport* shares most of these behaviors. We'll note some differences as we go along, but for the most, what can be done with `LOAD DATA` can be done with *mysqlimport* as well.

LOAD DATA provides options to address many of the import issues mentioned in the chapter introduction, such as the line-ending sequence for recognizing how to break input into records, the column value delimiter that permits records to be broken into separate values, the quoting character that may enclose column values, quoting and escaping conventions within values, and NULL value representation:

- By default, LOAD DATA expects the datafile to have the same number of columns as the table into which you load it, with the columns present in the same order as in the table. If the file column number or order differ from the table, you can specify which columns are present and their order. If the datafile contains fewer columns than the table, MySQL assigns default values for the missing columns.
- LOAD DATA assumes that data values are separated by tab characters and that lines end with linefeeds (newlines). If a file doesn't conform to these conventions, you can specify its format explicitly.
- You can indicate that data values may have quotes around them that should be stripped, and you can specify the quote character.
- Several special escape sequences are recognized and converted during input processing. The default escape character is backslash (\), but you can change it. The \N sequence is interpreted as a NULL value. The \b, \n, \r, \t, \\, and \0 sequences are interpreted as backspace, linefeed, carriage return, tab, backslash, and ASCII NUL characters. (NUL is a zero-valued byte; it differs from the SQL NULL value.)
- LOAD DATA provides diagnostic information about which input values cause problems. To display this information, execute a SHOW WARNINGS statement after the LOAD DATA statement.

The remainder of this section describes how to handle these issues using LOAD DATA or *mysqlimport*. It's lengthy because there's a lot to cover.

Specifying the datafile location

You can load files located either on the server host, or on the client host from which you issue the LOAD DATA statement. Telling MySQL where to find your datafile is a matter of knowing the rules that determine where it looks for the file (particularly important for files not in your current directory).

By default, the MySQL server assumes that the datafile is located on the server host. You can load local files that are located on the client host using LOAD DATA LOCAL rather than LOAD DATA, unless LOCAL capability is disabled by default. You might be able to enable it using the `--local-infile` option for *mysql*. If that doesn't work, your server has been configured to prohibit LOAD DATA LOCAL.

Many of the examples in this chapter assume that LOCAL can be used. If that's not true for your system, adapt the examples: omit LOCAL from the statement, make sure that the file is located on the MySQL server host and readable to the server, and specify the file pathname using the following rules. For example, specify the full pathname.

If the LOAD DATA statement includes no LOCAL keyword, the MySQL server looks for the file on the server host using the following rules:

- Your MySQL account must have the FILE privilege, and the file to be loaded must be either located in the data directory for the default database or world readable.
- An absolute pathname fully specifies the location of the file in the filesystem and the server reads it from the given location.
- A relative pathname is interpreted two ways, depending on whether it has a single component or multiple components. For a single-component filename such as *mytbl.txt*, the server looks for the file in the database directory for the default database. (The operation fails if you have not selected a default database.) For a multiple-component filename such as *xyz/mytbl.txt*, the server looks for the file beginning in the MySQL data directory. That is, it expects to find *mytbl.txt* in a directory named *xyz*.

Database directories are located directly under the server's data directory, so these two statements are equivalent if the default database is cookbook:

```
mysql> LOAD DATA INFILE 'mytbl.txt' INTO TABLE mytbl;
mysql> LOAD DATA INFILE 'cookbook/mytbl.txt' INTO TABLE mytbl;
```

If the LOAD DATA statement includes the LOCAL keyword, your client program reads the file on the client host and sends its contents to the server. The client interprets the pathname like this:

- An absolute pathname fully specifies the location of the file in the filesystem.
- A relative pathname specifies the file location relative to your current directory.

If your file is located on the client host, but you forget to indicate that it's local, an error occurs:

```
mysql> LOAD DATA 'mytbl.txt' INTO TABLE mytbl;
ERROR 1045 (28000): Access denied for user: 'user_name@host_name'
(Using password: YES)
```

That Access denied message can be confusing: if you're able to connect to the server and issue the LOAD DATA statement, it would seem that you've already gained access to MySQL, right? The error message means the server (not the client) tried to open *mytbl.txt* on the server host and could not access it.

If your MySQL server runs on the host from which you issue the LOAD DATA statement, "remote" and "local" refer to the same host. But the rules just discussed for locating datafiles still apply. Without LOCAL, the server reads the datafile directly. With LOCAL, the client program reads the file and sends its contents to the server.

mysqlimport uses the same rules for finding files as LOAD DATA. By default, it assumes that the datafile is located on the server host. To indicate that the file is local to the client host, specify the `--local` (or `-L`) option on the command line.

LOAD DATA assumes that the table is located in the default database. To load a file into a specific database, qualify the table name with the database name. The following statement indicates that the `mytbl` table is located in the `other_db` database:

```
mysql> LOAD DATA LOCAL 'mytbl.txt' INTO TABLE other_db.mytbl;
```

mysqlimport always requires a database argument:

```
% mysqlimport --local cookbook mytbl.txt
```

LOAD DATA assumes no relationship between the name of the datafile and the name of the table into which you load the file's contents. *mysqlimport* assumes a fixed relationship between the datafile name and the table name. Specifically, it uses the last component of the filename to determine the table name. For example, *mysqlimport* interprets *mytbl*, *mytbl.dat*, */home/paul/mytbl.csv*, and *C:\projects\mytbl.txt* all as files containing data for the `mytbl` table.

Naming Datafiles Under Windows

Windows systems use \ as the pathname separator in filenames. That's a bit of a problem because MySQL interprets backslash as the escape character in string values. To specify a Windows pathname, use either doubled backslashes or forward slashes. These two statements show two ways of referring to the same Windows file:

```
mysql> LOAD DATA LOCAL INFILE 'C:\\projects\\mydata.txt' INTO mytbl;
mysql> LOAD DATA LOCAL INFILE 'C:/projects/mydata.txt' INTO mytbl;
```

If the NO_BACKSLASH_ESCAPES SQL mode is enabled, backslash is not special, and you do not double it:

```
mysql> SET sql_mode = 'NO_BACKSLASH_ESCAPES';
mysql> LOAD DATA LOCAL INFILE 'C:\projects\mydata.txt' INTO mytbl;
```

Specifying column and line delimiters

By default, LOAD DATA assumes that datafile lines are terminated by linefeed (newline) characters and that values within a line are separated by tab characters. To provide explicit information about datafile format, use a FIELDS clause to describe the characteristics of fields within a line, and a LINES clause to specify the line-ending sequence.

The following LOAD DATA statement indicates that the input file contains data values separated by colons and lines terminated by carriage returns:

```
mysql> LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
    -> FIELDS TERMINATED BY ':' LINES TERMINATED BY '\r';
```

Each clause follows the table name. If both are present, FIELDS must precede LINES. The line and field termination indicators can contain multiple characters. For example, \r \n indicates that lines are terminated by carriage return/linefeed pairs.

The LINES clause also has a STARTING BY subclause. It specifies the sequence to be stripped from each input record. (Everything *up to* the given sequence is stripped. If you specify STARTING BY 'X' and a record begins with abcX, all four leading characters are stripped.) Like TERMINATED BY, the sequence can have multiple characters. If TERMINATED BY and STARTING BY both are present in the LINES clause, they can appear in any order.

For *mysqlimport*, command options provide the format specifiers. Commands that correspond to the preceding two LOAD DATA statements look like this:

```
% mysqlimport --local cookbook mytbl.txt
% mysqlimport --local --fields-terminated-by=":" --lines-terminated-by="\r" \
    cookbook mytbl.txt
```

Option order doesn't matter for *mysqlimport*.

The FIELDS and LINES clauses understand hex notation to specify arbitrary format characters, which is useful for loading datafiles that use binary format codes. Suppose that a datafile has lines with Ctrl-A between fields and Ctrl-B at the end of lines. The ASCII values for Ctrl-A and Ctrl-B are 1 and 2, so you represent them as 0x01 and 0x02:

```
FIELDS TERMINATED BY 0x01 LINES TERMINATED BY 0x02
```

mysqlimport also understands hex constants for format specifiers. You may find this capability helpful if you don't like remembering how to type escape sequences on the command line or when it's necessary to use quotes around them. Tab is 0x09, linefeed is 0x0a, and carriage return is 0x0d. This command indicates that the datafile contains tab-delimited lines terminated by CRLF pairs:

```
% mysqlimport --local --fields-terminated-by=0x09 \
    --lines-terminated-by=0x0d0a cookbook mytbl.txt
```

When you import datafiles, don't assume that LOAD DATA (or *mysqlimport*) knows more than it does. Some LOAD DATA frustrations occur because people expect MySQL to know more than it possibly can. Keep in mind that LOAD DATA has no idea at all about the format of your datafile. It makes certain assumptions about the input structure, represented as the default settings for the line and field terminators, and for the quote and escape character settings. If your input differs from those assumptions, you must tell MySQL so.

The line-ending sequence used in a datafile typically is determined by the system from which the file originated. Unix files normally have lines terminated by linefeeds, which you indicate like this:

```
LINES TERMINATED BY '\n'
```

Because `\n` happens to be the default line terminator, you need not specify that clause in this case unless you want to indicate the line-ending sequence explicitly. If files on your system don't use the Unix default (linefeed), you must specify the line terminator explicitly. For files that have lines ending in carriage returns or carriage return/linefeed pairs, respectively, use the appropriate `LINES TERMINATED BY` clause:

```
LINES TERMINATED BY '\r'
LINES TERMINATED BY '\r\n'
```

For example, to load a Windows file that contains tab-delimited fields and lines ending with CRLF pairs, use this `LOAD DATA` statement:

```
mysql> LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
    -> LINES TERMINATED BY '\r\n';
```

The corresponding *mysqlimport* command is:

```
% mysqlimport --local --lines-terminated-by="\r\n" cookbook mytbl.txt
```

If the file has been transferred from one machine to another, its contents may have been changed in subtle ways of which you're not aware. For example, an FTP transfer between machines running different operating systems typically translates line endings to those that are appropriate for the destination machine if the transfer is performed in text mode rather than in binary (image) mode.

When in doubt, check the contents of your datafile using a hex dump program or other utility that displays a visible representation of whitespace characters like tab, carriage return, and linefeed. Under Unix, programs such as *od* or *hexdump* can display file contents in a variety of formats. If you don't have these or some comparable utility, the *transfer* directory of the `recipes` distribution contains hex dumpers written in Perl, Ruby, and Python (*hexdump.pl*, *hexdump.rb*, and *hexdump.py*), as well as programs that display printable representations of all characters of a file (*see.pl*, *see.rb*, and *see.py*). You may find them useful for examining files to see what they really contain.

Dealing with quotes and special characters

If your datafile contains quoted values or escaped characters, tell `LOAD DATA` to be aware of them so that it doesn't load uninterpreted data values into the database.

The `FIELDS` clause can specify other format options besides `TERMINATED BY`. By default, `LOAD DATA` assumes that values are unquoted, and it interprets the backslash (`\`) as an escape character for special characters. To indicate the value-quoting character explic-

itly, use ENCLOSED BY; MySQL will strip that character from the ends of data values during input processing. To change the default escape character, use ESCAPED BY.

The three subclauses of the FIELDS clause (ENCLOSED BY, ESCAPED BY, and TERMINATED BY) may be present in any order if you specify more than one of them. For example, these FIELDS clauses are equivalent:

```
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
FIELDS ENCLOSED BY '"' TERMINATED BY ','
```

The TERMINATED BY value can consist of multiple characters. If data values are separated within input lines by *@*, sequences, indicate that like this:

```
FIELDS TERMINATED BY '*@*'
```

To disable escape processing entirely, specify an empty escape sequence:

```
FIELDS ESCAPED BY ''
```

When you specify ENCLOSED BY to indicate which quote character should be stripped from data values, it's possible to include the quote character literally within data values by doubling it or by preceding it with the escape character. For example, if the quote and escape characters are " and \, the input value "a""b\"c" is interpreted as a"b"c.

For *mysqlimport*, the corresponding command options for specifying quote and escape values are --fields-enclosed-by and --fields-escaped-by. (When using *mysqlimport* options that include quotes or backslashes or other characters that are special to your command interpreter, you may need to quote or escape the quote or escape characters.)

Handling duplicate key values

By default, an error occurs if an input record duplicates an existing row in the column or columns that form a PRIMARY KEY or UNIQUE index. To control this behavior, specify IGNORE or REPLACE after the filename to tell MySQL to either ignore duplicate rows or replace old rows with the new ones.

Suppose that you periodically receive meteorological data about current weather conditions from various monitoring stations, and that you store various measurements from these stations in a table that looks like this:

```
CREATE TABLE weatherdata
(
  station INT UNSIGNED NOT NULL,
  type    ENUM('precip','temp','cloudiness','humidity','barometer') NOT NULL,
  value   FLOAT,
  PRIMARY KEY (station, type)
);
```

The table includes a primary key on the combination of station ID and measurement type, to ensure that it contains only one row per station per type of measurement. The

table is intended to hold only current conditions, so when new measurements for a given station are loaded into the table, they should kick out the station's previous measurements. To accomplish this, use the REPLACE keyword:

```
mysql> LOAD DATA LOCAL INFILE 'data.txt' REPLACE INTO TABLE weatherdata;
```

mysqlimport has `--ignore` and `--replace` options that correspond to the IGNORE and REPLACE keywords for LOAD DATA.

Obtaining diagnostics about bad input data

LOAD DATA displays an information line to indicate whether there are any problematic input values. If so, use SHOW WARNINGS to find where they are and what the problems are.

When a LOAD DATA statement finishes, it returns a line of information that tells you how many errors or data conversion problems occurred. For example:

```
Records: 134  Deleted: 0  Skipped: 2  Warnings: 13
```

These values provide general information about the import operation:

- `Records` indicates the number of records found in the file.
- `Deleted` and `Skipped` are related to treatment of input records that duplicate existing table rows on unique index values. `Deleted` indicates how many rows were deleted from the table and replaced by input records, and `Skipped` indicates how many input records were ignored in favor of existing rows.
- `Warnings` is something of a catchall that indicates the number of problems found while loading data values into columns. Either a value stores into a column properly, or it doesn't. In the latter case, the value ends up in MySQL as something different, and MySQL counts it as a warning. (Storing a string `abc` into a numeric column results in a stored value of `0`, for example.)

What do these values tell you? The `Records` value normally should match the number of lines in the input file. If it doesn't, that's a sign that MySQL interprets the file as having a different format than it actually has. In this case, you'll likely also see a high `Warnings` value, which indicates that many values had to be converted because they didn't match the expected data type. The solution to this problem often is to specify the proper FIELDS and LINES clauses.

Assuming that your FIELDS and LINES format specifiers are correct, a nonzero `Warnings` count indicates the presence of bad input values. You can't tell from the numbers in the LOAD DATA information line which input records had problems or which columns were bad. To get that information, issue a SHOW WARNINGS statement.

Suppose that a table `t` has this structure:

```
CREATE TABLE t
(
```

```
  i INT,
  c CHAR(3),
  d DATE
);
```

And suppose that a datafile *data.txt* looks like this:

```
1           1           1
abc         abc         abc
2010-10-10  2010-10-10  2010-10-10
```

Loading the file into the table causes a number, a string, and a date to be loaded into each of the three columns. Doing so results in several data conversions and warnings, which you can see using `SHOW WARNINGS` immediately following `LOAD DATA`:

```
mysql> LOAD DATA LOCAL INFILE 'data.txt' INTO TABLE t;
Query OK, 3 rows affected, 5 warnings (0.01 sec)
Records: 3  Deleted: 0  Skipped: 0  Warnings: 5
mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------------+
| Level   | Code | Message                                             |
+---------+------+-----------------------------------------------------+
| Warning | 1265 | Data truncated for column 'd' at row 1              |
| Warning | 1366 | Incorrect integer value: 'abc' for column 'i' at row 2 |
| Warning | 1265 | Data truncated for column 'd' at row 2              |
| Warning | 1265 | Data truncated for column 'i' at row 3              |
| Warning | 1265 | Data truncated for column 'c' at row 3              |
+---------+------+-----------------------------------------------------+
5 rows in set (0.00 sec)
```

The `SHOW WARNINGS` output helps you determine which values were converted and why. The resulting table looks like this:

```
mysql> SELECT * FROM t;
+------+------+------------+
| i    | c    | d          |
+------+------+------------+
|    1 | 1    | 0000-00-00 |
|    0 | abc  | 0000-00-00 |
| 2010 | 201  | 2010-10-10 |
+------+------+------------+
```

Skipping datafile lines

To skip the first *n* lines of a datafile, add an `IGNORE` *n* `LINES` clause to the `LOAD DATA` statement. For example, a file might include an initial line of column labels. You can skip it like this:

```
mysql> LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
    -> IGNORE 1 LINES;
```

mysqlimport supports an `--ignore-lines=`*n* option that corresponds to `IGNORE` *n* `LINES`.

Specifying input column order

`LOAD DATA` assumes that columns in the datafile have the same order as the columns in the table. If that's not true, specify a list to indicate the table columns into which to load the datafile columns. Suppose that your table has columns `a`, `b`, and `c`, but successive columns in the datafile correspond to columns `b`, `c`, and `a`. Load the file like this:

```
mysql> LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl (b,c,a);
```

mysqlimport has a corresponding `--columns` option to specify the column list:

```
% mysqlimport --local --columns=b,c,a cookbook mytbl.txt
```

Preprocessing input values before inserting them

`LOAD DATA` can perform limited preprocessing of input values before inserting them, which sometimes enables you to map input data onto more appropriate values before loading them into your table. This is useful when values are not in a format suitable for loading into a table (for example, they are in the wrong units, or two input fields must be combined and inserted into a single column).

The previous section shows how to specify a column list for `LOAD DATA` to indicate how input fields correspond to table columns. The column list also can name user-defined variables, such that for each input record, the input fields are assigned to the variables. You can then perform calculations with those variables before inserting the result into the table. Specify these calculations in a `SET` clause that names one or more *`col_name`* = *`expr`* assignments, separated by commas.

Suppose that a datafile has the following columns, with the first line providing column labels:

```
Date        Time        Name        Weight      State
2006-09-01  12:00:00    Bill Wills  200         Nevada
2006-09-02  09:00:00    Jeff Deft   150         Oklahoma
2006-09-04  03:00:00    Bob Hobbs   225         Utah
2006-09-07  08:00:00    Hank Banks  175         Texas
```

Suppose also that the file is to be loaded into a table that has these columns:

```
CREATE TABLE t
(
  dt         DATETIME,
  last_name  CHAR(10),
  first_name CHAR(10),
  weight_kg  FLOAT,
  st_abbrev  CHAR(2)
);
```

To import the file, you must address several mismatches between its fields and the table columns:

- The file contains separate date and time fields that must be combined into date-and-time values for insertion into the DATETIME column.
- The file contains a name field, which must be split into separate first and last name values for insertion into the first_name and last_name columns.
- The file contains a weight in pounds, which must be converted to kilograms for insertion into the weight_kg column. (1 lb. equals .454 kg.)
- The file contains state names, but the table contains two-letter abbreviations. The name can be mapped to the abbreviation by performing a lookup in the states table.

To handle these conversions, skip the first line that contains the column labels, assign each input column to a user-defined variable, and write a SET clause to perform the calculations:

```
mysql> LOAD DATA LOCAL INFILE 'data.txt' INTO TABLE t
    -> IGNORE 1 LINES
    -> (@date,@time,@name,@weight_lb,@state)
    -> SET dt = CONCAT(@date,' ',@time),
    ->     first_name = SUBSTRING_INDEX(@name,' ',1),
    ->     last_name = SUBSTRING_INDEX(@name,' ',-1),
    ->     weight_kg = @weight_lb * .454,
    ->     st_abbrev = (SELECT abbrev FROM states WHERE name = @state);
```

After the import operation, the table contains these rows:

```
mysql> SELECT * FROM t;
+---------------------+-----------+------------+-----------+-----------+
| dt                  | last_name | first_name | weight_kg | st_abbrev |
+---------------------+-----------+------------+-----------+-----------+
| 2006-09-01 12:00:00 | Wills     | Bill       |      90.8 | NV        |
| 2006-09-02 09:00:00 | Deft      | Jeff       |      68.1 | OK        |
| 2006-09-04 03:00:00 | Hobbs     | Bob        |    102.15 | UT        |
| 2006-09-07 08:00:00 | Banks     | Hank       |     79.45 | TX        |
+---------------------+-----------+------------+-----------+-----------+
```

LOAD DATA can perform data value reformatting, as just shown. Other examples showing uses for this capability occur elsewhere. (For example, Recipe 11.4 uses it to map NULL values, and Recipe 12.13 rewrites non-ISO dates to ISO format during data import.) However, although LOAD DATA can map input values to other values, it cannot outright reject an input record that is found to contain unsuitable values. To do that, either preprocess the input file to remove these records or issue a DELETE statement after loading the file.

Ignoring datafile columns

Extra columns at the end of input lines are easy to handle. If a line contains more columns than are in the table, `LOAD DATA` just ignores them (although it might produce a nonzero warning count).

Skipping columns in the middle of lines is a bit more involved. To handle this, use a column list with `LOAD DATA` that assigns the columns to be ignored to a dummy user-defined variable. Suppose that you want to load information from a Unix password file */etc/passwd*, which contains lines in the following format:

```
account:password:UID:GID:GECOS:directory:shell
```

Suppose also that you don't want to load the password and directory columns. A table to hold the information in the remaining columns looks like this:

```
CREATE TABLE passwd
(
  account   CHAR(8),  # login name
  uid       INT,      # user ID
  gid       INT,      # group ID
  gecos     CHAR(60), # name, phone, office, etc.
  shell     CHAR(60)  # command interpreter
);
```

To load the file, specify that the column delimiter is a colon. Also, tell `LOAD DATA` to skip the second and sixth fields that contain the password and directory. To do this, add a column list in the statement. The list should include the name of each column to load into the table, and a dummy user-defined variable for columns to be ignored (you can use the same variable for all of them). The resulting statement looks like this:

```
mysql> LOAD DATA LOCAL INFILE '/etc/passwd' INTO TABLE passwd
    -> FIELDS TERMINATED BY ':'
    -> (account,@dummy,uid,gid,gecos,@dummy,shell);
```

The corresponding *mysqlimport* command includes a `--columns` option:

```
% mysqlimport --local \
    --columns="account,@dummy,uid,gid,gecos,@dummy,shell" \
    --fields-terminated-by=":" cookbook /etc/passwd
```

See Also

Another approach to ignoring columns is to preprocess the input file to remove columns. Recipe 11.7 discusses a *yank_col.pl* utility that can extract and display datafile columns in any order.

11.2. Importing CSV Files

Problem

You want to load a file that is in CSV format.

Solution

Use the appropriate format specifiers with `LOAD DATA` or *mysqlimport*.

Discussion

Datafiles in CSV format contain values that are delimited by commas rather than tabs and that may be quoted with double-quote characters. A CSV file *mytbl.txt* containing lines that end with carriage return/linefeed pairs can be loaded into `mytbl` using `LOAD DATA`:

```
mysql> LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES TERMINATED BY '\r\n';
```

Or like this using *mysqlimport*:

```
% mysqlimport --local --lines-terminated-by="\r\n" \
    --fields-terminated-by="," --fields-enclosed-by="\"" \
    cookbook mytbl.txt
```

11.3. Exporting Query Results from MySQL

Problem

You want to export the result of a query from MySQL into a file or another program.

Solution

Use the `SELECT ... INTO OUTFILE` statement, or redirect the output of the *mysql* program.

Discussion

The `SELECT ... INTO OUTFILE` statement exports a query result directly into a file on the server host. To capture the result on the client host instead, redirect the output of the *mysql* program. These methods have different strengths and weaknesses; get to know them both and apply whichever one best suits a given situation.

Exporting using the SELECT ... INTO OUTFILE statement

The syntax for this statement combines a regular `SELECT` with `INTO OUTFILE` *`file_name`*. The default output format is the same as for `LOAD DATA`, so the following statement exports the `passwd` table into */tmp/passwd.txt* as a tab-delimited, linefeed-terminated file:

```
mysql> SELECT * FROM passwd INTO OUTFILE '/tmp/passwd.txt';
```

To change the output format, use options similar to those used with `LOAD DATA` that indicate how to quote and delimit columns and records. For example, to export the `passwd` table (created earlier in Recipe 11.1) in CSV format with CRLF-terminated lines, use this statement:

```
mysql> SELECT * FROM passwd INTO OUTFILE '/tmp/passwd.txt'
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES TERMINATED BY '\r\n';
```

`SELECT ... INTO OUTFILE` has these properties:

- The output file is created directly by the MySQL server, so the filename should indicate where to write the file on the server host. The file location is determined using the same rules as for `LOAD DATA` without `LOCAL`, as described in Recipe 11.1. (There is no `LOCAL` version of the statement analogous to the `LOCAL` version of `LOAD DATA`.)
- You must have the MySQL `FILE` privilege to execute the `SELECT ... INTO OUTFILE` statement.
- The output file must not already exist. (This prevents MySQL from overwriting files that may be important.)
- You should have a login account on the server host or some way to access files on that host. `SELECT ... INTO OUTFILE` is of no value to you if you cannot retrieve the output file.
- Under Unix, the file is created world readable and is owned by the account used for running the MySQL server. This means that although you can read the file, you may not be able to delete it unless you can log in using that account.

Exporting using the mysql client program

Because `SELECT ... INTO OUTFILE` writes the datafile on the server host, you cannot use it unless your MySQL account has the `FILE` privilege. To export data into a local file owned by yourself, use another strategy. If all you require is tab-delimited output, do a "poor-man's export" by executing a `SELECT` statement with the *mysql* program and redirecting the output to a file. That way you can write query results into a file on your local host without the `FILE` privilege. Here's an example that exports the login name and command interpreter columns from the `passwd` table:

```
% mysql -e "SELECT account, shell FROM passwd" --skip-column-names \
    cookbook > shells.txt
```

The `-e` option specifies the statement to execute (see Recipe 1.5), and `--skip-column-names` tells MySQL not to write the row of column names that normally precedes statement output (see Recipe 1.7).

Note that MySQL writes `NULL` values as the string "NULL". Some postprocessing to convert them may be needed, depending on what you want to do with the output file.

It's possible to produce output in formats other than tab-delimited by sending the query result into a postprocessing filter that converts tabs to something else. For example, to use hash marks as delimiters, convert all tabs to # characters (*TAB* indicates where you type a tab character in the command):

```
% mysql --skip-column-names -e "your statement here" db_name \
    | sed -e "s/TAB/#/g" > output_file
```

You can also use *tr* for this purpose, although the syntax varies for different implementations of this utility. For Mac OS X or Linux, the command looks like this:

```
% mysql --skip-column-names -e "your statement here" db_name \
    | tr "\t" "#" > output_file
```

The *mysql* commands just shown use `--skip-column-names` to suppress column labels from appearing in the output. Under some circumstances, it may be useful to include the labels. (For example, if they will useful when importing the file later.) In that case, omit the `--skip-column-names` option from the command. In this respect, exporting query results with *mysql* is more flexible than `SELECT ... INTO OUTFILE` because the latter cannot produce output that includes column labels.

See Also

Another way to export query results to a file on the client host is to use the *mysql_to_text.pl* utility described in Recipe 11.5. That program has options that enable you to specify the output format explicitly. To export a query result as an Excel spreadsheet or XML document, see Recipes 11.8 and 11.9.

11.4. Importing and Exporting NULL Values

Problem

You need to represent `NULL` values in a datafile.

Solution

Use a value not otherwise present, so that you can distinguish NULL from all other legitimate non-NULL values. When you import the file, convert instances of that value to NULL.

Discussion

There's no standard for representing NULL values in datafiles, which makes them problematic for import and export operations. The difficulty arises from the fact that NULL indicates the *absence* of a value, and that's not easy to represent literally in a datafile. Using an empty column value is the most obvious thing to do, but that's ambiguous for string-valued columns because there is no way to distinguish a NULL represented that way from a true empty string. Empty values can be a problem for other data types as well. For example, if you load an empty value with LOAD DATA into a numeric column, it is stored as 0 rather than as NULL and thus becomes indistinguishable from a true 0 in the input.

The usual solution to this problem is to represent NULL using a value not otherwise present in the data. This is how LOAD DATA and *mysqlimport* handle the issue: they understand the value of \N by convention to mean NULL. (\N is interpreted as NULL only when it occurs by itself, not as part of a larger value such as x\N or \Nx.) For example, if you load the following datafile with LOAD DATA, it treats the instances of \N as NULL:

```
str1    13      1997-10-14
str2    \N      2009-05-07
\N      15      \N
\N      \N     1973-07-14
```

But you might want to interpret values other than \N as signifying NULL, and you might have different conventions in different columns. Consider the following datafile:

```
str1    13  1997-10-14
str2    -1  2009-05-07
Unknown 15
Unknown -1  1973-07-15
```

The first column contains strings, and Unknown signifies NULL. The second column contains integers, and -1 signifies NULL. The third column contains dates, and an empty value signifies NULL. What to do?

To handle situations like this, use LOAD DATA's input preprocessing capability: specify a column list that assigns input values to user-defined variables and use a SET clause that maps the special values to true NULL values. If the datafile is named *has_nulls.txt*, the following LOAD DATA statement properly interprets its contents:

```
mysql> LOAD DATA LOCAL INFILE 'has_nulls.txt'
    -> INTO TABLE t (@c1,@c2,@c3)
    -> SET c1 = IF(@c1='Unknown',NULL,@c1),
```

```
    ->     c2 = IF(@c2=-1,NULL,@c2),
    ->     c3 = IF(@c3='',NULL,@c3);
```

The resulting data after import looks like this:

```
+------+------+------------+
| c1   | c2   | c3         |
+------+------+------------+
| str1 |   13 | 1997-10-14 |
| str2 | NULL | 2009-05-07 |
| NULL |   15 | NULL       |
| NULL | NULL | 1973-07-15 |
+------+------+------------+
```

The preceding discussion pertains to interpreting `NULL` values for import into MySQL, but it's also necessary to think about `NULL` values when transferring data in the other direction—from MySQL into other programs. Here are some examples:

- `SELECT ... INTO OUTFILE` writes `NULL` values as `\N`. Will another program understand that convention? If not, convert `\N` to something the program understands. For example, the `SELECT` statement can export the column using an expression like this:

  ```
  IFNULL(col_name,'Unknown')
  ```

- You can use *mysql* in batch mode as an easy way to produce tab-delimited output (see Recipe 11.3), but then `NULL` values appear in the output as instances of the word "NULL". If that word occurs nowhere else in the output, you may be able to post-process it to convert instances of it to something more appropriate. For example, you can use a one-line *sed* command:

  ```
  % sed -e "s/NULL/\\N/g" data.txt > tmp
  ```

  If the word "NULL" appears where it represents something other than a `NULL` value, it's ambiguous and you should probably export your data differently. For example, use `IFNULL()` to map `NULL` values to something else.

11.5. Writing Your Own Data Export Programs

Problem

MySQL's built-in export capabilities don't suffice.

Solution

Write your own utilities.

Discussion

When existing export software doesn't do what you want, write your own programs. This section describes a Perl script, *mysql_to_text.pl*, that executes an arbitrary statement and exports it in the format you specify. It writes output to the client host and can include a row of column labels (two things that `SELECT ... INTO OUTFILE` cannot do). It produces multiple output formats more easily than by using *mysql* with a postprocessor, and it writes to the client host, unlike *mysqldump*, which can write only SQL-format output to the client. You can find *mysql_to_text.pl* in the *transfer* directory of the `recipes` distribution.

mysql_to_text.pl is based on the Text::CSV_XS module, which you must install on your system if it hasn't been already. To read its documentation, use this command:

```
% perldoc Text::CSV_XS
```

This module is convenient because it makes conversion of query output to CSV format relatively trivial. Your script need only provide an array of values, and the module packages them into a properly formatted output line. This makes it relatively trivial to convert query output to CSV format. But the real benefit of Text::CSV_XS is that it's configurable; you can tell it what delimiter and quote characters to use. This means that although the module produces CSV format by default, you can configure it to write a variety of output formats. For example, if you set the delimiter to tab and the quote character to `undef`, Text::CSV_XS generates tab-delimited output. We'll take advantage of that flexibility in this section for writing *mysql_to_text.pl*, and in Recipe 11.6 to write *cvt_file.pl*, a utility that converts files from one format to another.

mysql_to_text.pl accepts several command-line options. Some are used for specifying MySQL connection parameters (such as `--user`, `--password`, and `--host`). You're already familiar with these because they're used by the standard MySQL clients like *mysql*. The script also can obtain connection parameters from an option file, if you specify a `[client]` group in the file. In addition, *mysql_to_text.pl* accepts the following options:

`--execute=`*`query`*`, -e` *`query`*
: Execute *`query`* and export its output.

`--table=`*`tbl_name`*`, -t` *`tbl_name`*
: Export the contents of the named table. This is equivalent to using `--execute` to specify a *`query`* value of `SELECT * FROM` *`tbl_name`*.

`--labels`
: Include an initial row of column labels in the output

`--delim=`*`str`*
: Set the column delimiter to *`str`*. The option value can consist of one or more characters. The default is to use tabs.

`--quote=`*c*
: Set the column value quote character to *c*. The default is to not quote anything.

`--eol=`*str*
: Set the end-of-line sequence to *str*. The option value can consist of one or more characters. The default is to use linefeeds.

The defaults for the `--delim`, `--quote`, and `--eol` options correspond to those used by `LOAD DATA` and `SELECT ... INTO OUTFILE`.

The final argument on the command line should be the database name, unless it's implicit in the statement. For example, these two commands are equivalent; each exports the `passwd` table from the `cookbook` database in colon-delimited format:

```
% mysql_to_text.pl --delim=":" --table=passwd cookbook
% mysql_to_text.pl --delim=":" --table=cookbook.passwd
```

To generate CSV output with CRLF line terminators instead, use a command like this:

```
% mysql_to_text.pl --delim="," --quote="\"" --eol="\r\n" \
    --table=cookbook.passwd
```

That's a general description of how you use *mysql_to_text.pl*. Now let's discuss how it works. The initial part of the *mysql_to_text.pl* script declares a few variables, then processes the command-line arguments. As it happens, most of the code in the script is devoted to processing the command-line arguments and preparing to run the query. Very little of it involves interaction with MySQL:

```
#!/usr/bin/perl
# mysql_to_text.pl: Export MySQL query output in user-specified text format.

# Usage: mysql_to_text.pl [ options ] [db_name] > text_file

use strict;
use warnings;
use DBI;
use Text::CSV_XS;
use Getopt::Long;
$Getopt::Long::ignorecase = 0; # options are case sensitive
$Getopt::Long::bundling = 1;   # permit short options to be bundled

# ... construct usage message variable $usage (not shown) ...

# Variables for command line options - all undefined initially
# except for options that control output structure, which is set
# to be tab-delimited, linefeed-terminated.
my $help;
my ($host_name, $password, $port_num, $socket_name, $user_name, $db_name);
my ($stmt, $tbl_name);
my $labels;
my $delim = "\t";
my $quote;
```

```
my $eol = "\n";

GetOptions (
  # =i means an integer value is required after the option
  # =s means a string value is required after the option
  "help"          => \$help,        # print help message
  "host|h=s"      => \$host_name,   # server host
  "password|p=s"  => \$password,    # password
  "port|P=i"      => \$port_num,    # port number
  "socket|S=s"    => \$socket_name, # socket name
  "user|u=s"      => \$user_name,   # username
  "execute|e=s"   => \$stmt,        # statement to execute
  "table|t=s"     => \$tbl_name,    # table to export
  "labels|l"      => \$labels,      # generate row of column labels
  "delim=s"       => \$delim,       # column delimiter
  "quote=s"       => \$quote,       # column quoting character
  "eol=s"         => \$eol          # end-of-line (record) delimiter
) or die "$usage\n";

die "$usage\n" if defined ($help);

$db_name = shift (@ARGV) if @ARGV;

# One of --execute or --table must be specified, but not both
die "You must specify a query or a table name\n\n$usage\n"
  unless defined ($stmt) || defined ($tbl_name);
die "You cannot specify both a query and a table name\n\n$usage\n"
  if defined ($stmt) && defined ($tbl_name);

# interpret special chars in the file structure options
$quote = interpret_option ($quote);
$delim = interpret_option ($delim);
$eol = interpret_option ($eol);
```

The `interpret_option()` function (not shown) processes escape and hex sequences for the `--delim`, `--quote`, and `--eol` options. It interprets `\n`, `\r`, `\t`, and `\0` as linefeed, carriage return, tab, and the ASCII NUL character. It also interprets hex values, which can be given in `0x`*nn* form (for example, `0x0d` indicates a carriage return).

After processing the command-line options, *mysql_to_text.pl* constructs the data source name (DSN) and connects to the MySQL server:

```
my $dsn = "DBI:mysql:";
$dsn .= ";database=$db_name" if $db_name;
$dsn .= ";host=$host_name" if $host_name;
$dsn .= ";port=$port_num" if $port_num;
$dsn .= ";mysql_socket=$socket_name" if $socket_name;
# read [client] group parameters from standard option files
$dsn .= ";mysql_read_default_group=client";

my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
my $dbh = DBI->connect ($dsn, $user_name, $password, $conn_attrs);
```

The database name comes from the command line. Connection parameters can come from the command line or an option file. (Recipe 2.8 covers these option-processing techniques.)

After establishing a connection to MySQL, the script is ready to execute the query and produce output. This is where the Text::CSV_XS module comes into play. First, create a CSV object by calling `new()`, which takes an optional hash of options that control how the object handles data lines. The script prepares and executes the query, prints a row of column labels (if the `--labels` option was specified), and writes the rows of the result set:

```
my $csv = Text::CSV_XS->new ({
  sep_char    => $delim,
  quote_char  => $quote,
  escape_char => $quote,
  eol         => $eol,
  binary      => 1
});

# If table name was given, use it to create query that selects entire table.
# Split on dots in case it's a qualified name, to quote parts separately.
$stmt = "SELECT * FROM " . $dbh->quote_identifier (split (/\./, $tbl_name))
  if defined ($tbl_name);

warn "$stmt\n";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
if ($labels)                # write row of column labels
{
  $csv->combine (@{$sth->{NAME}}) or die "cannot process column labels\n";
  print $csv->string ();
}

my $count = 0;
while (my @val = $sth->fetchrow_array ())
{
  ++$count;
  $csv->combine (@val) or die "cannot process column values, row $count\n";
  print $csv->string ();
}
```

The `sep_char` and `quote_char` options in the `new()` call set the column delimiter and quoting character. The `escape_char` option is set to the same value as `quote_char` so that instances of the quote character occurring within data values are doubled in the output. The `eol` option indicates the line-termination sequence. Normally, Text::CSV_XS leaves it to you to print the terminator for output lines. By passing a non-`undef` `eol` value to `new()`, the module adds that value to every output line automatically. The `binary` option is useful for processing data values that contain binary characters.

After invoking `execute()`, the column labels are available in `$sth->{NAME}` (see Recipe 10.2). To produce each line of output, use `combine()` and `string()`. The `combine()` method takes an array of values and converts them to a properly formatted string. `string()` returns the string so we can print it.

11.6. Converting Datafiles from One Format to Another

Problem

You want to convert a file to a different format to make it easier to work with, or so that another program can understand it.

Solution

Use the *cvt_file.pl* conversion script described here.

Discussion

The *mysql_to_text.pl* script discussed in Recipe 11.5 uses MySQL as a data source and produces output in the format you specify via the `--delim`, `--quote`, and `--eol` options. This section describes *cvt_file.pl*, a utility that provides similar formatting options, but for both input and output. It reads data from a file rather than from MySQL, and converts it from one format to another. This enables the script to serve as a bridge between operations that use different formats. For example, invoke *cvt_file.pl* as follows to read a tab-delimited file *data.txt*, convert it to colon-delimited format, and write the result to *tmp.txt*:

```
% cvt_file.pl --idelim="\t" --odelim=":" data.txt > tmp.txt
```

The *cvt_file.pl* script has separate options for input and output. Thus, whereas *mysql_to_text.pl* has just a `--delim` option for specifying the column delimiter, *cvt_file.pl* has separate `--idelim` and `--odelim` options to set the input and output line column delimiters. But as a shortcut, `--delim` is also supported to set the delimiter for both input and output. The full set of options that *cvt_file.pl* understands is as follows:

`--idelim=`*str*, `--odelim=`*str*, `--delim=`*str*
: Set the column delimiter for input, output, or both. The option value can consist of one or more characters.

`--iquote=`*c*, `--oquote=`*c*, `--quote=`*c*
: Set the column quote character for input, output, or both.

`--ieol=`*str*, `--oeol=`*str*, `--eol=`*str*
: Set the end-of-line sequence for input, output, or both. The option value can consist of one or more characters.

`--iformat=`*`format`*, `--oformat=`*`format`*, `--format=`*`format`*
: Specify an input format, an output format, or both. This option is shorthand for setting the quote and delimiter values. `--iformat=csv` sets the input quote and delimiter characters to double quote and comma. `--iformat=tab` sets them to "no quotes" and tab.

`--ilabels`, `--olabels`, `--labels`
: Expect an initial line of column labels for input, write an initial line of labels for output, or both. If you request labels for the output but do not read labels from the input, *cvt_file.pl* uses column labels of `c1`, `c2`, and so forth.

cvt_file.pl assumes the same default file format as `LOAD DATA` and `SELECT INTO ... OUT FILE`, that is, tab-delimited lines terminated by linefeeds.

cvt_file.pl is located in the *transfer* directory of the `recipes` distribution. If you expect to use it regularly, install it in some directory that's listed in your search path so that you can invoke it from anywhere. Much of the source for the script is similar to *mysql_to_text.pl*, so rather than showing the code and discussing how it works, I'll just give some examples that illustrate how to use it:

- Read a file in CSV format with CRLF line termination, and write tab-delimited output with linefeed termination:

```
% cvt_file.pl --iformat=csv --ieol="\r\n" --oformat=tab --oeol="\n" \
    data.txt > tmp.txt
```

- Read and write CSV format, converting CRLF line terminators to carriage returns:

```
% cvt_file.pl --format=csv --ieol="\r\n" --oeol="\r" data.txt > tmp.txt
```

- Produce a tab-delimited file from the colon-delimited */etc/passwd* file:

```
% cvt_file.pl --idelim=":" /etc/passwd > tmp.txt
```

- Convert tab-delimited query output from `mysql` into CSV format:

```
% mysql -e "SELECT * FROM profile" cookbook \
    | cvt_file.pl --oformat=csv > profile.csv
```

11.7. Extracting and Rearranging Datafile Columns

Problem

You want to pull out only some columns from a datafile or rearrange them into a different order.

Solution

Use a utility that can produce columns from a file on demand.

Discussion

cvt_file.pl (see Recipe 11.6) serves as a tool that converts entire files from one format to another. Another common datafile operation is to manipulate columns. This is necessary, for example, when importing a file into a program that doesn't understand how to extract or rearrange input columns for itself. To work around this problem, rearrange the datafile instead.

Recall that this chapter began with a description of a scenario involving a 12-column CSV file *somedata.csv* from which only columns 2, 11, 5, and 9 were needed. To convert the file to tab-delimited format, do this:

```
% cvt_file.pl --iformat=csv somedata.csv > somedata.txt
```

But then what? If you just want to knock out a short script to extract those specific four columns, that's fairly easy: write a loop that reads input lines and writes only the desired columns, in the proper order. But that would be a special-purpose script, useful only within a highly limited context. With just a little more effort, it's possible to write a more general utility *yank_col.pl* that enables you to extract any set of columns. With such a tool, you specify the column list on the command line like this:

```
% yank_col.pl --columns=2,11,5,9 somedata.txt > tmp.txt
```

Because the script doesn't use a hardcoded column list, it can be used to extract an arbitrary set of columns in any order. Columns can be specified as a comma-separated list of column numbers or column ranges. (For example, `--columns=1,10,4-7` means columns 1, 10, 4, 5, 6, and 7.) *yank_col.pl* looks like this:

```
#!/usr/bin/perl
# yank_col.pl: Extract columns from input.

# Example: yank_col.pl --columns=2,11,5,9 filename

# Assumes tab-delimited, linefeed-terminated input lines.

# ... process command-line options (not shown) ...
# ... to get column list into @col_list array ...

while (<>)              # read input
{
  chomp;
  my @val = split (/\t/, $_, 10000);  # split, preserving all fields
  # extract desired columns, mapping undef to empty string (can
  # occur if an index exceeds number of columns present in line)
  @val = map { defined ($_) ? $_ : "" } @val[@col_list];
  print join ("\t", @val) . "\n";
}
```

The input processing loop converts each line to an array of values, then pulls out from the array the values corresponding to the requested columns. To avoid looping through

the array, it uses Perl's notation that permits a list of subscripts to be specified all at once to request multiple array elements. For example, if `@col_list` contains the values 2, 6, and 3, these two expressions are equivalent:

```
($val[2] , $val[6], $val[3])
@val[@col_list]
```

What if you want to extract columns from a file that's not in tab-delimited format, or produce output in another format? In that case, combine *yank_col.pl* with the *cvt_file.pl* script. Suppose that you want to pull out all but the password column from the colon-delimited */etc/passwd* file and write the result in CSV format. Use *cvt_file.pl* both to preprocess */etc/passwd* into tab-delimited format for *yank_col.pl* and to postprocess the extracted columns into CSV format:

```
% cvt_file.pl --idelim=":" /etc/passwd \
    | yank_col.pl --columns=1,3-7 \
    | cvt_file.pl --oformat=csv > passwd.csv
```

To avoid typing all of that as one long command, use temporary files for the intermediate steps:

```
% cvt_file.pl --idelim=":" /etc/passwd > tmp1.txt
% yank_col.pl --columns=1,3-7 tmp1.txt > tmp2.txt
% cvt_file.pl --oformat=csv tmp2.txt > passwd.csv
% rm tmp1.txt tmp2.txt
```

Forcing split() to Return Every Field

The Perl `split()` function is extremely useful, but normally omits trailing empty fields. This means that if you write only as many fields as `split()` returns, output lines may not have the same number of fields as input lines. To avoid this problem, pass a third argument to indicate the maximum number of fields to return. This forces `split()` to return as many fields as are actually present on the line or the number requested, whichever is smaller. If the value of the third argument is large enough, the practical effect is to cause all fields to be returned, empty or not. Scripts shown in this chapter use a field count value of 10,000:

```
# split line at tabs, preserving all fields
my @val = split (/\t/, $_, 10000);
```

In the (unlikely?) event that an input line has more fields than that, it is truncated. If you think that will be a problem, bump up the number even higher.

11.8. Exchanging Data Between MySQL and Microsoft Excel

Problem

You want to exchange information between MySQL and Excel.

Solution

Your programming language might provide library routines to make this task easier. For example, you can use Perl modules that read and write Excel spreadsheet files to construct data transfer utilities.

Discussion

If you need to transfer Excel files into MySQL, check around for modules that let you do this from your chosen programming language. For example, the following modules enable reading and writing Excel spreadsheets in Perl scripts:

- Spreadsheet::ParseExcel::Simple provides an easy-to-use interface for reading Excel spreadsheets. (Because Microsoft occasionally revises spreadsheet formats, you might need to save a spreadsheet in an older format so that this module can read it.)
- Excel::Writer::XLSX enables you to create files in Excel spreadsheet format.

These Excel modules are available from the Perl CPAN. (They're actually frontends to other modules, which you must also install as prerequisites.) After installing the modules, use these commands to read their documentation:

```
% perldoc Spreadsheet::ParseExcel::Simple
% perldoc Excel::Writer::XLSX
```

These modules make it relatively easy to write short scripts for converting spreadsheets to and from tab-delimited file format. Combined with techniques for importing and exporting data to and from MySQL, these scripts can help you move spreadsheet contents to MySQL tables and vice versa. Use them as is, or adapt them to suit your own purposes.

The following script, *from_excel.pl*, reads an Excel spreadsheet and converts it to tab-delimited format:

```
#!/usr/bin/perl
# from_excel.pl: Read Excel spreadsheet, write tab-delimited,
# linefeed-terminated output to the standard output.

use strict;
```

```
use warnings;
use Spreadsheet::ParseExcel::Simple;

@ARGV or die "Usage: $0 excel-file\n";

my $xls = Spreadsheet::ParseExcel::Simple->read ($ARGV[0]);
foreach my $sheet ($xls->sheets ())
{
  while ($sheet->has_data ())
  {
    my @data = $sheet->next_row ();
    print join ("\t", @data) . "\n";
  }
}
```

The *to_excel.pl* script performs the converse operation of reading a tab-delimited file and writing it in Excel format:

```
#!/usr/bin/perl
# to_excel.pl: Read tab-delimited, linefeed-terminated input, write
# Excel-format output to the standard output.

use strict;
use warnings;
use Excel::Writer::XLSX;

binmode (STDOUT);
my $ss = Excel::Writer::XLSX->new (\*STDOUT);
my $ws = $ss->add_worksheet ();
my $row = 0;

while (<>)                              # read each row of input
{
  chomp;
  my @data = split (/\t/, $_, 10000); # split, preserving all fields
  my $col = 0;
  foreach my $val (@data)               # write row to the worksheet
  {
    $ws->write ($row, $col, $val);
    $col++;
  }
  $row++;
}
```

to_excel.pl assumes input in tab-delimited, linefeed-terminated format. Use it in conjunction with *cvt_file.pl* (see Recipe 11.6) to work with files not in that format.

Another Excel-related Perl module, Spreadsheet::WriteExcel::FromDB, reads data from a table using a DBI connection and writes it in Excel format. Here's a script that exports a MySQL table as an Excel spreadsheet:

```
#!/usr/bin/perl
# mysql_to_excel.pl: Given a database and table name,
```

```
# dump the table to the standard output in Excel format.

use strict;
use warnings;
use DBI;
use Spreadsheet::ParseExcel::Simple;
use Spreadsheet::WriteExcel::FromDB;

# ... process command-line options (not shown) ...
# ... to get $db_name, $tbl_name ...
# ... connect to database (not shown) ...

my $ss = Spreadsheet::WriteExcel::FromDB->read ($dbh, $tbl_name);
binmode (STDOUT);
print $ss->as_xls ();
```

Each utility writes to its standard output, which you can redirect to capture the results in a file:

```
% from_excel.pl data.xls > data.txt
% to_excel.pl data.txt > data.xlsx
% mysql_to_excel.pl cookbook profile > profile.xls
```

Note that *from_excel.pl* and *mysql_to_excel.pl* read and write *.xls* files, whereas *to_excel.pl* writes *.xlsx* files.

See Also

On Windows, MySQL for Excel is an add-in that enables access to MySQL databases directly from Excel. For information, visit the "Download MySQL for Excel" page on the MySQL website.

11.9. Exporting Query Results as XML

Problem

You want to export the result of a query as an XML document.

Solution

mysql can do that, or you can write your own exporter.

Discussion

The *mysql* client can produce XML-format output from a query result (see Recipe 1.7). You can also write your own XML-export programs. One way to do this is to execute a query and then write the result, adding the XML markup yourself. Another is to install a few Perl modules and let them do the work:

- XML::Generator::DBI executes a query over a DBI connection and passes the result to a suitable output writer.
- XML::Handler::YAWriter provides one such writer.

The following script, *mysql_to_xml.pl*, is somewhat similar to *mysql_to_text.pl* (see Recipe 11.5), but doesn't take options for such things as the quote or delimiter characters. They are unneeded for writing XML because the XML writer module handles those issues. *mysql_to_xml.pl* understands these options:

`--execute=`*`query`*, `-e` *`query`*
: Execute *`query`* and export its output.

`--table=`*`tbl_name`*, `-t` *`tbl_name`*
: Export the contents of the named table. This is equivalent to using `--execute` to specify a *`query`* value of `SELECT * FROM` *`tbl_name`*.

If necessary, you can also specify standard connection parameter options such as `--user` or `--host`. The final argument on the command line should be the database name, unless it's implicit in the query.

Suppose that a table named `expt` contains test scores from an experiment:

```
mysql> SELECT * FROM expt;
+---------+------+-------+
| subject | test | score |
+---------+------+-------+
| Jane    | A    |    47 |
| Jane    | B    |    50 |
| Jane    | C    |  NULL |
| Jane    | D    |  NULL |
| Marvin  | A    |    52 |
| Marvin  | B    |    45 |
| Marvin  | C    |    53 |
| Marvin  | D    |  NULL |
+---------+------+-------+
```

To export the contents of `expt`, invoke *mysql_to_xml.pl* using either of the following commands:

```
% mysql_to_xml.pl --execute="SELECT * FROM expt" cookbook > expt.xml
% mysql_to_xml.pl --table=cookbook.expt > expt.xml
```

The resulting XML document, *expt.xml*, looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rowset>
 <select query="SELECT * FROM expt">
  <row>
   <subject>Jane</subject>
   <test>A</test>
   <score>47</score>
```

```
  </row>
  <row>
   <subject>Jane</subject>
   <test>B</test>
   <score>50</score>
  </row>
…
  <row>
   <subject>Marvin</subject>
   <test>C</test>
   <score>53</score>
  </row>
  <row>
   <subject>Marvin</subject>
   <test>D</test>
  </row>
 </select>
</rowset>
```

Each table row is written as a `<row>` element. Within a row, column names and values are used as element names and values, one element per column. Note that `NULL` values are omitted from the output.

The script produces this output with very little code after it processes the command-line arguments and connects to the MySQL server. The XML-related parts of *mysql_to_xml.pl* are the `use` statements that pull in the necessary modules and the code to set up and use the XML objects. Given a database handle `$dbh` and a query string `$query`, the code instructs the writer object to send its results to the standard output, then connects that object to DBI and issues the query:

```
#!/usr/bin/perl
# mysql_to_xml.pl: Given a database and table name,
# dump the table to the standard output in XML format.

use strict;
use warnings;
use DBI;
use XML::Generator::DBI;
use XML::Handler::YAWriter;

# ... process command-line options (not shown) ...
# ... connect to database (not shown) ...

# Create output writer; "-" means "standard output"
my $out = XML::Handler::YAWriter->new (AsFile => "-");
# Set up connection between DBI and output writer
my $gen = XML::Generator::DBI->new (
  dbh         => $dbh,    # database handle
  Handler     => $out,    # output writer
  RootElement => "rowset" # document root element
);
```

```
# If table name was given, use it to create query that selects entire table.
# Split on dots in case it's a qualified name, to quote parts separately.
$stmt = "SELECT * FROM " . $dbh->quote_identifier (split (/\./, $tbl_name))
  if defined ($tbl_name);

# Issue query and write XML
$gen->execute ($stmt);

$dbh->disconnect ();
```

Other languages might have library modules to perform similar XML export operations. For example, the Ruby DBI::Utils::XMLFormatter module has a `table` method that exports a query result as XML. Here's a simple script that uses it:

```
#!/usr/bin/ruby -w
# xmlformatter.rb: Demonstrate DBI::Utils::XMLFormatter.table method.

require "Cookbook"

stmt = "SELECT * FROM expt"
# override statement with command line argument if one was given
stmt = ARGV[0] if ARGV.length > 0

dbh = Cookbook.connect
DBI::Utils::XMLFormatter.table(dbh.select_all(stmt))
dbh.disconnect
```

11.10. Importing XML into MySQL

Problem

You want to import an XML document into a MySQL table.

Solution

Set up an XML parser to read the document, then use the document records to construct and execute `INSERT` statements.

Discussion

Importing an XML document depends on being able to parse the document and extract record contents from it. How you do that depends on how the document is written. For example, one format might represent column names and values as attributes of `<column>` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<rowset>
  <row>
```

```
    <column name="subject" value="Jane" />
    <column name="test" value="A" />
    <column name="score" value="47" />
   </row>
   <row>
    <column name="subject" value="Jane" />
    <column name="test" value="B /> 
    <column name="score" value="50" />
   </row>
…
</rowset>
```

Another format uses column names as element names and column values as the contents of those elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<rowset>
  <row>
   <subject>Jane</subject>
   <test>A</test>
   <score>47</score>
  </row>
  <row>
   <subject>Jane</subject>
   <test>B</test>
   <score>50</score>
  </row>
…
</rowset>
```

Due to the various structuring possibilities, it's necessary to make some assumptions about the format you expect the XML document to have. For the example here, I assume the second format just shown. One way to process this kind of document is to use the XML::XPath module, which enables you to refer to elements within the document using path expressions. For example, the path `//row` selects all the `<row>` elements under the document root, and the path `*` selects all child elements of a given element. You can use these paths with XML::XPath to obtain first a list of all the `<row>` elements, and then for each row a list of all its columns.

The following script, *xml_to_mysql.pl*, takes three arguments:

```
% xml_to_mysql.pl db_name tbl_name xml_file
```

The filename argument indicates which document to import, and the database and table name arguments indicate the table into which to import it. *xml_to_mysql.pl* processes the command-line arguments, connects to MySQL, and processes the document:

```
#!/usr/bin/perl
# xml_to_mysql.pl: Read XML file into MySQL.

use strict;
use warnings;
```

```
use DBI;
use XML::XPath;

# ... process command-line options (not shown) ...
# ... connect to database (not shown) ...

# Open file for reading
my $xp = XML::XPath->new (filename => $file_name);
my $row_list = $xp->find ("//row");     # find set of <row> elements
print "Number of records: " . $row_list->size () . "\n";
foreach my $row ($row_list->get_nodelist ())    # loop through rows
{
  my @name; # array for column names
  my @val;  # array for column values
  my $col_list = $row->find ("*");              # child columns of row
  foreach my $col ($col_list->get_nodelist ())  # loop through columns
  {
    push (@name, $col->getName ());             # save column name
    push (@val, $col->string_value ());         # save column value
  }
  # construct INSERT statement, then execute it
  my $stmt = "INSERT INTO $tbl_name ("
             . join (",", @name)
             . ") VALUES ("
             . join (",", ("?") x scalar (@val))
             . ")";
  $dbh->do ($stmt, undef, @val);
}

$dbh->disconnect ();
```

The script creates an `XML::XPath` object, which opens and parses the document. This object is queried for the set of `<row>` elements, using the path `//row`. The size of this set indicates how many rows the document contains.

To process each row, the script uses the path `*` to ask for all the child elements of the row object. Each child corresponds to a column within the row; using `*` as the path for `get_nodelist()` this way is convenient because you need not know in advance which columns to expect. *xml_to_mysql.pl* obtains the name and value from each column and saves them in the `@name` and `@value` arrays. After all the columns have been extracted, the arrays are used to construct an `INSERT` statement that names those columns that were found to be present in the row and that includes a placeholder for each data value. (Recipe 2.5 discusses placeholder list construction.) Then the script executes the statement, passing the column values to `do()` to bind them to the placeholders.

In Recipe 11.9, we used *mysql_to_xml.pl* to export the contents of the `expt` table as an XML document. *xml_to_mysql.pl* performs the converse operation of importing the document back into MySQL:

```
% xml_to_mysql.pl cookbook expt expt.xml
```

As it processes the document, the script generates and executes the following set of statements:

```
INSERT INTO expt (subject,test,score) VALUES ('Jane','A','47')
INSERT INTO expt (subject,test,score) VALUES ('Jane','B','50')
INSERT INTO expt (subject,test) VALUES ('Jane','C')
INSERT INTO expt (subject,test) VALUES ('Jane','D')
INSERT INTO expt (subject,test,score) VALUES ('Marvin','A','52')
INSERT INTO expt (subject,test,score) VALUES ('Marvin','B','45')
INSERT INTO expt (subject,test,score) VALUES ('Marvin','C','53')
INSERT INTO expt (subject,test) VALUES ('Marvin','D')
```

Note that these statements do not all insert the same number of columns. MySQL will set the missing columns to their default values.

11.11. Guessing Table Structure from a Datafile

Problem

Someone gives you a datafile and says, "Here, put this into MySQL for me." But no table yet exists to hold the data.

Solution

Use a utility that guesses the table structure by examining the datafile contents.

Discussion

Sometimes you must import data into MySQL for which no table has yet been set up. You can create the table yourself, based on any knowledge you have about the contents of the file. Or you might be able to avoid some of the work by using *guess_table.pl*, a utility located in the *transfer* directory of the `recipes` distribution. *guess_table.pl* reads the datafile to see what kind of information it contains, then attempts to produce an appropriate `CREATE TABLE` statement that matches the contents of the file. This script is necessarily imperfect because column contents sometimes are ambiguous. (For example, a column containing a small number of distinct strings might be a `VARCHAR` column or an `ENUM`.) Still, it may be easier to tweak the `CREATE TABLE` statement that *guess_table.pl* produces than to write the statement from scratch. This utility also has diagnostic value, although that's not its primary purpose. For example, if you believe a column contains only numbers, but *guess_table.pl* indicates that it should be a `VARCHAR` column, that tells you the column contains at least one nonnumeric value.

guess_table.pl assumes that its input is in tab-delimited, linefeed-terminated format. It also assumes valid input because any attempt to guess data types based on possibly flawed data is doomed to failure. This means, for example, that if a date column is to be recognized as such, it should be in ISO format. Otherwise, *guess_table.pl* may charac-

terize it as a VARCHAR column. If a datafile doesn't satisfy these assumptions, you may be able to reformat it first using the *cvt_file.pl* and *cvt_date.pl* utilities described in Recipes 11.6 and 12.12.

guess_table.pl understands the following options:

`--labels`
: Interpret the first input line as a row of column labels and use them for table column names. Without this option, *guess_table.pl* uses default column names of `c1`, `c2`, and so forth.

  If the file contains a row of labels and you omit this option, *guess_table.pl* treats the labels as data values. The likely result is that the script will characterize *all* columns as VARCHAR columns (even those that otherwise contain only numeric or temporal values), due to the presence of a nonnumeric or nontemporal value in the column.

`--lower`, `--upper`
: Force column names in the CREATE TABLE statement to be lowercase or uppercase.

`--quote-names`, `--skip-quote-names`
: Quote or do not quote table and column identifiers in the CREATE TABLE statement with ` characters (for example, `` `mytbl` ``). This can be useful if an identifier is a reserved word. The default is to quote identifiers.

`--report`
: Generate a report rather than a CREATE TABLE statement. The script displays the information that it gathers about each column.

`--table=`*`tbl_name`*
: Specify the table name to use in the CREATE TABLE statement. The default name is `t`.

Here's an example of how *guess_table.pl* works. Suppose that a file named *commodities.csv* is in CSV format and has the following contents:

```
commodity,trade_date,shares,price,change
sugar,12-14-2014,1000000,10.50,-.125
oil,12-14-2014,96000,60.25,.25
wheat,12-14-2014,2500000,8.75,0
gold,12-14-2014,13000,103.25,2.25
sugar,12-15-2014,970000,10.60,.1
oil,12-15-2014,105000,60.5,.25
wheat,12-15-2014,2370000,8.65,-.1
gold,12-15-2014,11000,101,-2.25
```

The first row indicates the column labels, and the following rows contain data records, one per line. The values in the `trade_date` column are dates, but they are in *`MM-DD-CCYY`* format rather than the ISO format that MySQL expects. *cvt_date.pl* can convert these dates to ISO format. However, both *cvt_date.pl* and *guess_table.pl* require input

in tab-delimited, linefeed-terminated format, so first use *cvt_file.pl* to convert the input to tab-delimited, linefeed-terminated format, and *cvt_date.pl* to convert the dates:

```
% cvt_file.pl --iformat=csv commodities.csv > tmp1.txt
% cvt_date.pl --iformat=us tmp1.txt > tmp2.txt
```

Feed the resulting file, *tmp2.txt*, to *guess_table.pl*:

```
% guess_table.pl --labels --table=commodities tmp2.txt > commodities.sql
```

The `CREATE TABLE` statement that *guess_table.pl* writes to *commodities.sql* looks like this:

```
CREATE TABLE `commodities`
(
  `commodity` VARCHAR(5) NOT NULL,
  `trade_date` DATE NOT NULL,
  `shares` BIGINT UNSIGNED NOT NULL,
  `price` DOUBLE UNSIGNED NOT NULL,
  `change` DOUBLE NOT NULL
);
```

guess_table.pl produces that statement based on heuristics such as these:

- A column that contains only numeric values is assumed to be a `BIGINT` if no values contain a decimal point, and `DOUBLE` otherwise.
- A numeric column that contains no negative values is likely to be `UNSIGNED`.
- If a column contains no empty values, *guess_table.pl* assumes that it's probably `NOT NULL`.
- Columns that cannot be classified as numbers or dates are taken to be `VARCHAR` columns, with a length equal to the longest value present in the column.

You might want to edit the `CREATE TABLE` statement that *guess_table.pl* produces, to make modifications such as using smaller integer types, increasing the size of character fields, changing `VARCHAR` to `CHAR`, adding indexes, or changing a column name that is a reserved word in MySQL.

To create the table, use the statement produced by *guess_table.pl*:

```
% mysql cookbook < commodities.sql
```

Then load the datafile into the table (skipping the initial row of labels):

```
mysql> LOAD DATA LOCAL INFILE 'tmp2.txt' INTO TABLE commodities
    -> IGNORE 1 LINES;
```

The resulting table contents after import look like this:

```
mysql> SELECT * FROM commodities;
+-----------+------------+---------+--------+--------+
| commodity | trade_date | shares  | price  | change |
+-----------+------------+---------+--------+--------+
| sugar     | 2014-12-14 | 1000000 |   10.5 | -0.125 |
```

```
| oil       | 2014-12-14 |   96000 |  60.25 |   0.25 |
| wheat     | 2014-12-14 | 2500000 |   8.75 |      0 |
| gold      | 2014-12-14 |   13000 | 103.25 |   2.25 |
| sugar     | 2014-12-15 |  970000 |   10.6 |    0.1 |
| oil       | 2014-12-15 |  105000 |   60.5 |   0.25 |
| wheat     | 2014-12-15 | 2370000 |   8.65 |   -0.1 |
| gold      | 2014-12-15 |   11000 |    101 |  -2.25 |
+-----------+------------+---------+--------+--------+
```

CHAPTER 12
Validating and Reformatting Data

12.0. Introduction

The previous chapter, Chapter 11, focuses on methods for moving data into and out of MySQL. The present chapter is related in that it also covers data transfer issues, but here the emphasis is on issues of datafile *content* rather than *structure*. For example, if you don't know whether the values contained in a file are legal, preprocess it to check or reformat them. Numeric values may need verification as lying within a specific range, dates may need conversion to or from ISO format, and so forth.

The chapter deals with formatting and validation issues primarily within the context of checking entire files, but many of the techniques discussed here can be applied in other situations as well. Consider a web-based application that presents a form for a user to fill in and then processes its contents to create a new row in the database. Web APIs generally make form contents available as a set of already parsed discrete values, so the application may not need to deal with record and column delimiters. On the other hand, validation issues remain paramount. You really have no idea what kind of values a user is sending your script, so it's important to check them. This chapter covers validation extensively, and Recipe 20.6 revisits the issue in web context.

For additional background on the material covered here, see the introduction to Chapter 11.

Source code for program fragments and scripts discussed in this chapter is located in the *transfer* directory of the `recipes` distribution, with the exception that some utility functions are contained in library files located in the *lib* directory.

12.1. Using the SQL Mode to Reject Bad Input Values

Problem

By default, MySQL is forgiving about accepting data values that are invalid, out of range, or otherwise unsuitable for the data types of the columns into which you insert them. But you want the server to be more restrictive and not accept bad data.

Solution

Set the SQL mode. Several mode values are available to control how strict the server is. Some modes apply generally to all input values. Others apply to specific data types such as dates.

Discussion

Normally, MySQL coerces input values to the data types of your table columns if the input doesn't match. Consider the following table, which has integer, string, and date columns:

```
mysql> CREATE TABLE t (i INT, c CHAR(6), d DATE);
```

Inserting a row with unsuitable data values into the table causes warnings (which you can see with `SHOW WARNINGS`), but the server loads the values into the table after coercing them to some value that fits the column:

```
mysql> INSERT INTO t (i,c,d) VALUES('-1x','too-long string!','1999-02-31');
mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1265 | Data truncated for column 'i' at row 1    |
| Warning | 1265 | Data truncated for column 'c' at row 1    |
| Warning | 1264 | Out of range value for column 'd' at row 1 |
+---------+------+-------------------------------------------+
mysql> SELECT * FROM t;
+------+--------+------------+
| i    | c      | d          |
+------+--------+------------+
|   -1 | too-lo | 0000-00-00 |
+------+--------+------------+
```

One way to prevent these warnings is to check the input data on the client side to make sure that it's legal. This is a reasonable strategy in certain circumstances (see the sidebar in Recipe 12.2), but there is an alternative: let the server check data values on the server side and reject them with an error if they're invalid.

To do this, set the `sql_mode` system variable to enable server restrictions on input data acceptance. With the proper restrictions in place, data values that would otherwise result

in conversions and warnings result in errors instead. Try the preceding INSERT again after enabling "strict" SQL mode:

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES';
mysql> INSERT INTO t (i,c,d) VALUES('-1x','too-long string!','1999-02-31');
ERROR 1265 (01000): Data truncated for column 'i' at row 1
```

Here the statement doesn't even progress to the second and third data values because the first is invalid for an integer column and the server raises an error.

Without input restrictions enabled, the server checks that the month part of date values is in the range from 1 to 12 and that the day value is legal for the given month. This means that `'2005-02-31'` generates a warning by default (with conversion to `'0000-00-00'`). In strict mode, an error occurs.

MySQL still permits dates such as `'1999-11-00'` or `'1999-00-00'` that have zero parts, or the "zero" date (`'0000-00-00'`), and (until MySQL 5.7.4) this is true even in strict mode. To restrict these kinds of date values, enable the NO_ZERO_IN_DATE and NO_ZERO_DATE SQL modes to cause warnings, or errors in strict mode. For example, to prohibit dates with zero parts or "zero" dates, set the SQL mode like this:

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE';
```

A simpler way to enable these restrictions, and a few more besides, is to enable TRADITIONAL SQL mode. TRADITIONAL mode is actually a constellation of modes, as you can see by setting and displaying the sql_mode value:

```
mysql> SET sql_mode = 'TRADITIONAL';
mysql> SELECT @@sql_mode\G
*************************** 1. row ***************************
@@sql_mode: STRICT_TRANS_TABLES,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,
            NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,TRADITIONAL,
            NO_AUTO_CREATE_USER
```

You can read more about the various SQL modes in the *MySQL Reference Manual*.

The examples shown set the session value of the sql_mode system variable, so they change the SQL mode only for your current session. To set the mode globally for all clients, start the server with a `--sql_mode=`*`mode_value`* option. Alternatively, if you have the SUPER privilege, you can set the global mode at runtime:

```
mysql> SET GLOBAL sql_mode = 'mode_value';
```

12.2. Validating and Transforming Data

Problem

You must make sure that the data values in a file are legal.

Solution

Check them, possibly rewriting them into a more suitable format.

Discussion

Recipes in Chapter 11 show how to work with the structural characteristics of files, by reading lines and breaking them into separate columns. But sometimes you must focus on the data content of a file, not only its structure:

- It's often a good idea to validate data values to make sure they're legal for the data types into which you store them. For example, you can make sure that values intended for `INT`, `DATE`, and `ENUM` columns are integers, dates in *`CCYY-MM-DD`* format, and legal enumeration values, respectively.
- Data values may need reformatting. You might store credit card values as a string of digits but permit users of a web application to separate blocks of digits by spaces or dashes. These values must be rewritten before storing them. Rewriting dates from one format to another is especially common; for example, if a program writes dates in *`MM-DD-YY`* format to ISO format for import into MySQL. If a program understands only date and time formats and not a combined date-and-time format (such as MySQL uses for the `DATETIME` and `TIMESTAMP` data types), you must split date-and-time values into separate date and time values.

This is the first of a set of recipes that describe validation and reformatting techniques that help in these kinds of situations. Techniques covered here for checking values include pattern matching and validation against information in a database. It's not unusual for certain validation operations to occur repeatedly, in which case you'll probably find it useful to construct a library of functions. By packaging validation operations as library routines, it is easier to write utilities based on them, and the utilities make it easier to perform command-line operations on entire files so that you can avoid editing them yourself.

To avoid writing your own library routines, look around to see if someone else has already written suitable routines that you can use. For example, if you check the Perl CPAN (*cpan.perl.org*), you'll find a Data::Validate module hierarchy. The modules there provide library routines that standardize a number of common validation tasks. Data::Validate::MySQL deals specifically with MySQL data types.

Server-Side Versus Client-Side Validation

As described in Recipe 12.1, you can cause data validation to be done on the server side by setting the SQL mode to be restrictive about accepting bad input data. In this case, the MySQL server raises an error for values that are invalid for the data types of the columns into which you insert them.

In the next few sections, the focus is validation on the client side rather than on the server side. Client-side validation can be useful when you require more control over validation than simply receiving an error from the server. (For example, if you test values yourself, it's often easier to provide more informative messages to users about the exact nature of problems with the values.) Also, it might be necessary to couple validation with reformatting to transform complex values so that they are compatible with MySQL data types. You have more flexibility to do this on the client side.

Writing an input-processing loop

Many of the validation recipes shown in the new few sections are typical of those that you perform within the context of a program that reads a file and checks individual column values. The general framework for such a file-processing utility looks like this:

```
#!/usr/bin/perl
# loop.pl: Typical input-processing loop.

# Assumes tab-delimited, linefeed-terminated input lines.

use strict;
use warnings;

while (<>)                # read each line
{
  chomp;
  # split line at tabs, preserving all fields
  my @val = split (/\t/, $_, 10000);
  for my $i (0 .. @val - 1) # iterate through fields in line
  {
    # ... test $val[$i] here ...
  }
}
```

The `while()` loop reads each input line. Within the loop, each line is broken into fields. (Recipe 11.7 discusses why `split()` is written as it is.) The inner `for()` loop iterates through the fields, enabling each to be processed in sequence. If you don't apply a given test uniformly to all the fields, replace the `for()` loop with separate column-specific tests.

This loop assumes tab-delimited, linefeed-terminated input, an assumption shared by most of the utilities discussed throughout this chapter. To use these utilities with datafiles in other formats, you may be able to convert such files to tab-delimited format using the *cvt_file.pl* script discussed in Recipe 11.6.

Putting common tests in libraries

It may be useful to package a test that you perform often as a library function. This makes the operation easy to perform and also gives it a name that's likely to make the meaning of the operation clearer than the comparison code itself. The following test performs a pattern match to check that `$val` consists entirely of digits (optionally preceded by a plus sign), and then makes sure the value is greater than zero:

```
$valid = ($val =~ /^\+?\d+$/ && $val > 0);
```

In other words, the test looks for strings that represent positive integers. To make the test easier to use and its intent clearer, package it as a function that is used like this:

```
$valid = is_positive_integer ($val);
```

Define the function as follows:

```
sub is_positive_integer
{
my $s = $_[0];

  return $s =~ /^\+?\d+$/ && $s > 0;
}
```

Now put the function definition into a library file so that multiple scripts can use it easily. The *Cookbook_Utils.pm* module file in the *lib* directory of the `recipes` distribution is an example of a library file that contains a number of validation functions. Take a look through it to see which functions may be useful in your own programs (or as a model for writing your own library files). To gain access to this module from within a script, include a `use` statement like this:

```
use Cookbook_Utils;
```

You must of course install the module file in a directory where Perl will find it (see Recipe 2.3).

A significant benefit of putting a collection of utility routines into a library file is that you can use it for all kinds of programs. It's rare for a data manipulation problem to be completely unique. If you can pick and choose at least a few validation routines from a library, it reduces the amount of code you must write, even for highly specialized programs.

12.3. Using Pattern Matching to Validate Data

Problem

You must compare a value to a set of values that is difficult to specify literally without writing a really ugly expression.

Solution

Use pattern matching.

Discussion

Pattern matching is a powerful validation tool that enables you to test entire classes of values with a single expression. You can also use pattern tests to break matched values into subparts for further individual testing or in substitution operations to rewrite matched values. For example, you might break a matched date into pieces to verify that the month is in the range from 1 to 12, and the day is within the number of days in the month. You might use a substitution to reorder *MM-DD-YY* or *DD-MM-YY* values into *YY-MM-DD* format.

The next few sections describe how to use patterns to test several types of values, but first let's review some general pattern-matching principles. The following discussion focuses on Perl's regular-expression capabilities. Pattern matching in Ruby, PHP, and Python is similar, although you should consult the relevant documentation for any differences. For Java, use the `java.util.regex` package.

In Perl, the pattern constructor is */pat/*:

```
$it_matched = ($val =~ /pat/);    # pattern match
```

Put an `i` after the */pat/* constructor to make the pattern match case insensitive:

```
$it_matched = ($val =~ /pat/i);   # case-insensitive match
```

To use a character other than slash, begin the constructor with `m`. This is useful if the pattern itself contains slashes:

```
$it_matched = ($val =~ m|pat|);   # alternate constructor character
```

To look for a nonmatch, replace the `=~` operator with the `!~` operator:

```
$no_match = ($val !~ /pat/);      # negated pattern match
```

To perform a substitution in `$val` based on a pattern match, use `s/`*pat*`/`*replacement*`/`. If *pat* occurs within `$val`, it's replaced by *replacement*. To perform a case-insensitive match, put an `i` after the last slash. To perform a global substitution that replaces all instances of *pat* rather than only the first one, add a `g` after the last slash:

```
$val =~ s/pat/replacement/;    # substitution
$val =~ s/pat/replacement/i;   # case-insensitive substitution
$val =~ s/pat/replacement/g;   # global substitution
$val =~ s/pat/replacement/ig;  # case-insensitive and global
```

The following table shows some of the special pattern elements available in Perl regular expressions:

Pattern	What the pattern matches
`^`	Beginning of string
`$`	End of string
`.`	Any character
`\s`, `\S`	Whitespace or nonwhitespace character
`\d`, `\D`	Digit or nondigit character
`\w`, `\W`	Word (alphanumeric or underscore) or nonword character
`[...]`	Any character listed between the square brackets
`[^...]`	Any character not listed between the square brackets
`p1``|`*`p2`*`|`*`p3`*	Alternation; matches any of the patterns *`p1`*, *`p2`*, or *`p3`*
`*`	Zero or more instances of preceding element
`+`	One or more instances of preceding element
`{`*`n`*`}`	*`n`* instances of preceding element
`{`*`m`*`,`*`n`*`}`	*`m`* through *`n`* instances of preceding element

Many of these pattern elements are the same as those available for MySQL's `REGEXP` regular-expression operator (see Recipe 5.9).

To match a literal instance of a character that is special within patterns, such as `*`, `^`, or `$`, precede it with a backslash. Similarly, to include a character within a character class construction that is special in character classes (`[`, `]`, or `-`), precede it with a backslash. To include a literal `^` in a character class, list it somewhere other than as the first character between the brackets.

Many of the validation patterns shown in the following sections are of the form `/^`*`pat`*`$/`. Beginning and ending a pattern with `^` and `$` has the effect of requiring *`pat`* to match the entire string that you test. This is common in data validation contexts because it's generally desirable to know that a pattern matches an entire input value, not only part of it. (To be sure that a value represents an integer, for example, it does no good to know only that it contains an integer somewhere.) This is not a hard-and-fast rule, however, and sometimes it's useful to perform a more relaxed test by omitting the `^` and `$` characters as appropriate. For example, if you want to strip leading and trailing whitespace from a value, use one pattern anchored only to the beginning of the string, and another anchored only to the end:

```
$val =~ s/^\s+//;   # trim leading whitespace
$val =~ s/\s+$//;   # trim trailing whitespace
```

That's such a common operation, in fact, that it's a good candidate for being written as a utility function. The *Cookbook_Utils.pm* file contains a function `trim_whitespace()` that performs both substitutions and returns the result:

```
$val = trim_whitespace ($val);
```

To remember subsections of a string matched by a pattern, use parentheses around the relevant pattern parts. After a successful match, you can refer to the matched substrings using the variables `$1`, `$2`, and so forth:

```
if ("2019-05-23" =~ /^(\d+)(.*)$/)
{
  $first_part = $1; # this is the year, 2019
  $the_rest = $2;   # this is the rest of the date
}
```

To indicate that an element within a pattern is optional, follow it with a `?` character. To match values consisting of a sequence of digits, optionally beginning with a minus sign, and optionally ending with a period, use this pattern:

```
/^-?\d+\.?$/
```

Use parentheses to group alternations within a pattern. The following pattern matches time values in *`hh:mm`* format, optionally followed by `AM` or `PM`:

```
/^\d{1,2}:\d{2}\s*(AM|PM)?$/i
```

The use of parentheses in that pattern also has the side effect of remembering the optional part in `$1`. To suppress that side effect, use `(?:`*`pat`*`)` instead:

```
/^\d{1,2}:\d{2}\s*(?:AM|PM)?$/i
```

That's sufficient background in Perl pattern matching to enable construction of useful validation tests for several types of data values. The following sections provide patterns that can be used to test for broad content types, numbers, temporal values, and email addresses or URLs.

The *transfer* directory of the `recipes` distribution contains a *test_pat.pl* script that reads input values, matches them against several patterns, and reports which patterns each value matches. The script is easily extensible, so you can use it as a test harness to try your own patterns.

12.4. Using Patterns to Match Broad Content Types

Problem

You want to classify values into broad categories.

Solution

Use a pattern that is similarly broad.

Discussion

To check whether values are empty or nonempty, or consist only of certain types of characters, the patterns listed in the following table may suffice:

Pattern	Type of value the pattern matches
`/^$/`	Empty value
`/./`	Nonempty value
`/^\s*$/`	Whitespace, possibly empty
`/^\s+$/`	Nonempty whitespace
`/\S/`	Nonempty, and not only whitespace
`/^\d+$/`	Digits only, nonempty
`/^[a-z]+$/i`	Alphabetic characters only (case insensitive), nonempty
`/^\w+$/`	Alphanumeric or underscore characters only, nonempty

12.5. Using Patterns to Match Numeric Values

Problem

You must make sure a string looks like a number.

Solution

Use a pattern that matches the type of number you're looking for.

Discussion

Patterns can be used to classify values into several types of numbers, as shown in the following table:

Pattern	Type of value the pattern matches
`/^\d+$/`	Unsigned integer
`/^-?\d+$/`	Negative or unsigned integer
`/^[-+]?\d+$/`	Signed or unsigned integer
`/^[-+]?(\d+(\.\d*)?|\.\d+)$/`	Floating-point number

The pattern `/^\d+$/` matches unsigned integers by requiring a nonempty value that consists only of digits from the beginning to the end of the value. If you care only that a value begins with an integer, you can match an initial numeric part and extract it. To

do this, match only the initial part of the string (omit the $ that requires the pattern to match to the end of the string) and place parentheses around the \d+ part. Then refer to the matched number as $1 after a successful match:

```
if ($val =~ /^(\d+)/)
{
  $val = $1;  # reset value to matched subpart
}
```

You could also add zero to the value, which causes Perl to perform an implicit string-to-number conversion that discards the nonnumeric suffix:

```
if ($val =~ /^\d+/)
{
  $val += 0;
}
```

However, this method of discarding trailing nonnumeric characters has a disadvantage: the conversion generates warnings for values that actually have a nonnumeric part if you run Perl with the -w option or include a use warnings line in your script (which I recommend). It also converts string values like 0013 to the number 13, which may be unacceptable in some contexts. See Recipe 2.4 for additional discussion and an alternative approach.

Some kinds of numeric values have a special format or other unusual constraints. Here are a few examples and how to deal with them:

ZIP codes

ZIP and ZIP+4 codes are postal codes used for mail delivery in the United States. They have values like 12345 or 12345-6789 (that is, five digits, possibly followed by a dash and four more digits). To match one form or the other, or both forms, use the patterns shown in the following table:

Pattern	Type of value the pattern matches
/^\d{5}$/	ZIP code, five digits only
/^\d{5}-\d{4}$/	ZIP+4 code
/^\d{5}(-\d{4})?$/	ZIP or ZIP+4 code

Credit card numbers

Credit card numbers typically consist of digits, but it's common for values to be written with spaces, dashes, or other characters between groups of digits. For example, the following numbers are equivalent:

```
0123456789012345
0123 4567 8901 2345
0123-4567-8901-2345
```

To match such values, use this pattern:

```
/^[- \d]+/
```

(Perl permits the `\d` digit specifier within character classes.) However, that pattern doesn't identify values of the wrong length, and it may be useful to remove extraneous characters before storing values in MySQL. To require credit card values to contain 16 digits, use a substitution that removes all nondigits, then check the length of the result:

```
$val =~ s/\D//g;
$valid = (length ($val) == 16);
```

12.6. Using Patterns to Match Dates or Times

Problem

You must make sure a string looks like a date or time.

Solution

Use a pattern that matches the type of temporal value you expect. Be sure to consider issues such as how strict to be about delimiters between subparts and the lengths of the subparts.

Discussion

Dates are a validation headache because they come in so many formats. Pattern tests are extremely useful for weeding out illegal values, but often insufficient for full verification: a date might have a number where you expect a month, but the date isn't valid if the number is 13. This section introduces some patterns that match a few common date formats. Recipe 12.11 revisits this topic in more detail and discusses combining pattern tests with content verification.

To require values to be dates in ISO (*CCYY-MM-DD*) format, use this pattern:

```
/^\d{4}-\d{2}-\d{2}$/
```

The pattern requires the `-` character as the delimiter between date parts. To permit either `-` or `/` as the delimiter, use a character class between the numeric parts (the slashes are escaped with a backslash to prevent them from being interpreted as the end of the pattern constructor):

```
/^\d{4}[-\/]\d{2}[-\/]\d{2}$/
```

To avoid the backslashes, use a different delimiter around the pattern:

```
m|^\d{4}[-/]\d{2}[-/]\d{2}$|
```

To permit any nondigit delimiter (which corresponds to how MySQL operates when it interprets strings as dates), use this pattern:

```
/^\d{4}\D\d{2}\D\d{2}$/
```

To permit leading zeros in values like `03` to be missing, just look for three nonempty digit sequences:

```
/^\d+\D\d+\D\d+$/
```

Of course, that pattern is so general that it also matches other values such as US Social Security numbers (which have the format 012-34-5678). To constrain the subpart lengths by requiring two to four digits in the year part and one or two digits in the month and day parts, use this pattern:

```
/^\d{2,4}?\D\d{1,2}\D\d{1,2}$/
```

For dates in other formats such as *MM-DD-YY* or *DD-MM-YY*, similar patterns apply, but the subparts are arranged in a different order. This pattern matches both of those formats:

```
/^\d{2}-\d{2}-\d{2}$/
```

To check the values of individual date parts, use parentheses in the pattern and extract the substrings after a successful match. If you expect dates to be in ISO format, for example, do this:

```
if ($val =~ /^(\d{2,4})\D(\d{1,2})\D(\d{1,2})$/)
{
  ($year, $month, $day) = ($1, $2, $3);
}
```

The library file *lib/Cookbook_Utils.pm* in the `recipes` distribution contains several of these pattern tests, packaged as function calls. If the date doesn't match the pattern, they return `undef`. Otherwise, they return a reference to an array containing the broken-out values for the year, month, and day. This can be useful for performing further checking on the components of the date. For example, `is_iso_date()` looks for dates that match ISO format. It's defined as follows:

```
sub is_iso_date
{
my $s = $_[0];

  return undef unless $s =~ /^(\d{2,4})\D(\d{1,2})\D(\d{1,2})$/;
  return [ $1, $2, $3 ];  # return year, month, day
}
```

Use the function like this:

```
my $ref = is_iso_date ($val);
if (defined ($ref))
{
  # $val matched ISO format pattern;
  # check its subparts using $ref->[0] through $ref->[2]
}
else
```

```
{
  # $val didn't match ISO format pattern
}
```

You'll often find additional processing necessary with dates because date-matching patterns help to weed out values that are syntactically malformed, but don't assess whether the individual components contain legal values. To do that, some range checking is necessary. Recipe 12.11 covers that topic.

If you're willing to skip subpart testing and just want to rewrite the pieces, use a substitution. For example, to rewrite values assumed to be in *MM-DD-YY* format into *YY-MM-DD* format, do this:

```
$val =~ s/^(\d+)\D(\d+)\D(\d+)$/$3-$1-$2/;
```

Time values are somewhat more orderly than dates, usually being written with hours first and seconds last, with two digits per part:

```
/^\d{2}:\d{2}:\d{2}$/
```

To be more lenient, permit the hours part to have a single digit, or the seconds part to be missing:

```
/^\d{1,2}:\d{2}(:\d{2})?$/
```

Mark parts of the time with parentheses if you want to range-check the individual parts, or perhaps to reformat the value to include a seconds part of `00` if it happens to be missing. However, this requires some care with the parentheses and the `?` characters in the pattern if the seconds part is optional. You want to permit the entire `:\d{2}` at the end of the pattern to be optional, but not to save the `:` character in `$3` if the third time section is present. To accomplish that, use `(?:`*pat*`)`, a grouping notation that doesn't save the matched substring. Within that notation, use parentheses around the digits to save them. Then `$3` is `undef` if the seconds part is not present, and contains the seconds digits otherwise:

```
if ($val =~ /^(\d{1,2}):(\d{2})(?::(\d{2}))?$/)
{
  my ($hour, $min, $sec) = ($1, $2, $3);
  $sec = "00" if !defined ($sec); # seconds missing; use 00
  $val = "$hour:$min:$sec";
}
```

To rewrite times from 12-hour format with AM and PM suffixes to 24-hour format, do this:

```
if ($val =~ /^(\d{1,2}):(\d{2})(?::(\d{2}))?\s*(AM|PM)?$/i)
{
  my ($hour, $min, $sec) = ($1, $2, $3);
  # supply missing seconds
  $sec = "00" unless defined ($sec);
  if ($hour == 12 && (!defined ($4) || uc ($4) eq "AM"))
  {
```

```
    $hour = "00"; # 12:xx:xx AM times are 00:xx:xx
  }
  elsif ($hour < 12 && defined ($4) && uc ($4) eq "PM")
  {
    $hour += 12;  # PM times other than 12:xx:xx
  }
  $val = "$hour:$min:$sec";
}
```

The time parts are placed into `$1`, `$2`, and `$3`, with `$3` set to `undef` if the seconds part is missing. The suffix goes into `$4` if it's present. If the suffix is `AM` or missing (`undef`), the value is interpreted as an AM time. If the suffix is `PM`, the value is interpreted as a PM time.

See Also

This section is just the beginning of what you can do when processing dates for data-transfer purposes. Date and time testing and conversion can be highly idiosyncratic, and the sheer number of issues to consider is mind-boggling:

- What is the basic date format? Dates come in several common styles, such as ISO (*`CCYY-MM-DD`*), US (*`MM-DD-YY`*), and British (*`DD-MM-YY`*) formats. And these are just some of the more standard formats. Many more are possible. For example, a datafile may contain dates written as `June 17, 1959` or as `17 Jun '59`.
- Are trailing times permitted on dates, or perhaps required? When times are expected, is the full time required or just the hour and minute?
- Do you permit special values like `now` or `today`?
- Are date parts required to be delimited by a particular character, such as `-` or `/`, or are other delimiters permitted?
- Are date parts required to have a specific number of digits? Or are leading zeros on month and year values permitted to be missing?
- Are months written numerically, or represented as month names like `January` or `Jan`?
- Are two-digit year values permitted? Should they be converted to have four digits? If so, what is the transition point within the range `00` to `99` at which values change from one century to another?
- Should date parts be checked to ensure their validity? Patterns can recognize strings that look like dates or times, but while they're extremely useful for detecting malformed values, they may not be sufficient. A value like `1947-15-99` may match a pattern but isn't a legal date. Pattern testing is thus most useful in conjunction with range checks on the individual parts of the date.

The prevalence of these issues in data-transfer problems means that you'll probably end up writing some of your own validators on occasion to handle very specific date formats. Other sections of this chapter can provide additional assistance. For example, Recipe 12.10 covers conversion of two-digit year values to four-digit form, and Recipe 12.11 discusses how to perform validity checking on components of date or time values.

You might be able to save yourself some work by using existing date-checking modules for your API language. Some possibilities: the Perl `Date` module; the Ruby `date` module; the Python `datetime` module; the PHP `DateTime` class; the Java `GregorianCalendar` and `SimpleDateTime` classes.

12.7. Using Patterns to Match Email Addresses or URLs

Problem

You want to determine whether a value looks like an email address or a URL.

Solution

Use a pattern, tuned to the desired level of strictness.

Discussion

The immediately preceding sections use patterns to identify classes of values such as numbers and dates, which are fairly typical applications for regular expressions. But pattern matching has much more widespread applicability for data validation. To give some idea of a few other types of values for which pattern matching can be used, this section shows a few tests for email addresses and URLs.

To check values that are expected to be email addresses, the pattern should require at least an `@` character with nonempty strings on either side:

```
/.@./
```

That's a pretty minimal test. It's difficult to come up with a fully general pattern that covers all the legal values and rejects all the illegal ones, but it's easy to write a pattern that's at least a little more restrictive. For example, in addition to being nonempty, the username and the domain name should consist entirely of characters other than `@` characters or spaces:

```
/^[^@ ]+@[^@ ]+$/
```

You may also want to require that the domain name part contain at least two parts separated by a dot:

```
/^[^@ ]+@[^@ .]+\.[^@ .]+/
```

To look for URL values that begin with a protocol specifier of `http://`, `ftp://`, or `mailto:`, use an alternation that matches any of them at the beginning of the string. These values contain slashes, so it's easier to use a different character around the pattern to avoid escaping the slashes with backslashes:

```
m#^(http://|ftp://|mailto:)#i
```

The alternatives in the pattern are grouped within parentheses because otherwise the `^` anchors only the first of them to the beginning of the string. The `i` modifier follows the pattern because protocol specifiers in URLs are not case sensitive. The pattern is otherwise fairly unrestrictive because it permits anything to follow the protocol specifier. Add further restrictions as necessary.

12.8. Using Table Metadata to Validate Data

Problem

You must check input values against the legal members of an `ENUM` or `SET` column.

Solution

Get the column definition, extract the list of members from it, and check data values against the list.

Discussion

Some forms of validation involve checking input values against information stored in a database. This includes values to be stored in an `ENUM` or `SET` column, which can be checked against the valid members stored in the column definition. Database-backed validation also applies to values that must match those listed in a lookup table to be considered legal. For example, input records that contain customer IDs can be required to match a row in a `customers` table, and state abbreviations in addresses can be verified against a table that lists each state. This recipe describes `ENUM`- and `SET`-based validation, and Recipe 12.9 discusses how to use lookup tables.

One way to check input values that correspond to the legal values of `ENUM` or `SET` columns is to get the list of legal column values into an array using the information in `INFORMATION_SCHEMA`, then perform an array membership test. For example, the favorite-color column `color` from the `profile` table is an `ENUM` defined as follows:

```
mysql> SELECT COLUMN_TYPE FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'profile'
    -> AND COLUMN_NAME = 'color';
```

```
+---------------------------------------------------+
| COLUMN_TYPE                                       |
+---------------------------------------------------+
| enum('blue','red','green','brown','black','white') |
+---------------------------------------------------+
```

If you extract the list of enumeration members from the `COLUMN_TYPE` value and store them in an array `@members`, you can perform the membership test like this:

```
$valid = grep (/^$val$/i, @members);
```

The pattern constructor begins and ends with `^` and `$` to require `$val` to match an entire enumeration member (rather than just a substring). It also is followed by an `i` to specify a case-insensitive comparison because the default collation is `latin1_swedish_ci`, which is case-insensitive. (If you have a column with a different collation, adjust accordingly.)

In Recipe 10.7, we wrote a function `get_enumorset_info()` that returns `ENUM` or `SET` column metadata. This includes the list of members, so it's easy to use that function to write another utility routine, `check_enum_value()`, that gets the legal enumeration values and performs the membership test. The routine takes four arguments: a database handle, the table name and column name for the `ENUM` column, and the value to check. It returns true or false to indicate whether the value is legal:

```
sub check_enum_value
{
my ($dbh, $db_name, $tbl_name, $col_name, $val) = @_;

  my $valid = 0;
  my $info = get_enumorset_info ($dbh, $db_name, $tbl_name, $col_name);
  if ($info && uc ($info->{type}) eq "ENUM")
  {
    # use case-insensitive comparison because default collation
    # (latin1_swedish_ci) is case-insensitive (adjust if you use
    # a different collation)
    $valid = grep (/^$val$/i, @{$info->{values}});
  }
  return $valid;
}
```

For single-value testing, such as to validate a value submitted in a web form, that kind of test works well. However, to test a lot of values (like an entire column in a datafile), it's better to read the enumeration values into memory once, then use them repeatedly to check each data value. Furthermore, it's a lot more efficient to perform hash lookups than array lookups (in Perl at least). To do so, retrieve the legal enumeration values and store them as keys of a hash. Then test each input value by checking whether it exists as a hash key. It's a little more effort to construct the hash, which is why `check_enum_value()` doesn't do so. But for bulk validation, the improved lookup speed more than makes up for the hash construction overhead. (To check for yourself the relative efficiency of

array membership tests versus hash lookups, try the *lookup_time.pl* script in the *transfer* directory of the `recipes` distribution.)

Begin by getting the metadata for the column and convert the list of legal enumeration members to a hash:

```
my $ref = get_enumorset_info ($dbh, $db_name, $tbl_name, $col_name);
my %members;
# convert hash key to consistent lettercase
map { $members{lc ($_)} = 1; } @{$ref->{values}};
```

The `map` expression makes each enumeration member exist as the key of a hash element. The hash key is what's important here; the value associated with it is irrelevant. (The example shown sets the value to `1`, but you could use `undef`, `0`, or any other value.) Note that the code converts the hash keys to lowercase before storing them. This is done because hash key lookups in Perl are case sensitive. That's fine if the values that you check also are case sensitive, but `ENUM` columns by default are not. By converting the enumeration values to a given lettercase before storing them in the hash, and then converting the values you want to check similarly, you perform, in effect, a case-insensitive key existence test:

```
$valid = exists ($members{lc ($val)});
```

The preceding example converts enumeration values and input values to lowercase. You could just as well use uppercase, as long as you do so for all values consistently.

Note that the existence test may fail if the input value is the empty string. You must decide how to handle that case on a column-by-column basis. For example, if the column permits `NULL` values, you might interpret the empty string as equivalent to `NULL` and thus as being a legal value.

The validation procedure for `SET` values is similar to that for `ENUM` values, except that an input value might consist of any number of `SET` members, separated by commas. For the value to be legal, each element in it must be legal. In addition, because "any number of members" includes "none," the empty string is a legal value for any `SET` column.

For one-shot testing of individual input values, use a utility routine `check_set_value()` that is similar to `check_enum_value()`:

```
sub check_set_value
{
my ($dbh, $db_name, $tbl_name, $col_name, $val) = @_;

  my $valid = 0;
  my $info = get_enumorset_info ($dbh, $db_name, $tbl_name, $col_name);
  if ($info && uc ($info->{type}) eq "SET")
  {
    return 1 if $val eq "";    # empty string is legal element
    # use case-insensitive comparison because default collation
    # (latin1_swedish_ci) is case-insensitive (adjust if you use
```

```
    # a different collation)
    $valid = 1;     # assume valid until we find out otherwise
    foreach my $v (split (/,/, $val))
    {
      if (!grep (/^$v$/i, @{$info->{values}}))
      {
        $valid = 0; # value contains an invalid element
        last;
      }
    }
  }
  return $valid;
}
```

For bulk testing, construct a hash from the legal `SET` members. The procedure is the same as shown previously for producing a hash from `ENUM` elements.

To validate a given input value against the `SET` member hash, convert it to the same lettercase as the hash keys, split it at commas to get a list of the individual elements of the value, and then check each one. If any of the elements are invalid, the entire value is invalid:

```
$valid = 1;     # assume valid until we find out otherwise
foreach my $elt (split (/,/, lc ($val)))
{
  if (!exists ($members{$elt}))
  {
    $valid = 0; # value contains an invalid element
    last;
  }
}
```

After the loop terminates, `$valid` is true if the value is legal for the `SET` column, and false otherwise. Empty strings are always legal `SET` values, but this code performs no special-case test for an empty string. No such test is necessary because in that case the `split()` operation returns an empty list, the loop never executes, and `$valid` remains true.

12.9. Using a Lookup Table to Validate Data

Problem

You must check values to make sure they're listed in a lookup table.

Solution

Issue statements to check whether the values are in the table. The best way to do this depends on the number of input values and the table size.

Discussion

To validate input values against the contents of a lookup table, the techniques are somewhat similar to those shown in Recipe 12.8 for checking `ENUM` and `SET` columns. However, whereas `ENUM` and `SET` columns usually have a small number of member values, a lookup table can have an essentially unlimited number of values. You might not want to read them all into memory.

Validation of input values against the contents of a lookup table can be done several ways, as illustrated in the following discussion. The tests shown in the examples perform comparisons against values exactly as they are stored in the lookup table. To perform case-insensitive comparisons, convert all values to a consistent lettercase. (See the discussion of case conversion in Recipe 12.8.)

Issue individual statements

For one-shot operations, test a value by checking whether it's listed in the lookup table. The following query returns true (nonzero) for a value that is present and false otherwise:

```
$valid = $dbh->selectrow_array (
           "SELECT COUNT(*) FROM $tbl_name WHERE val = ?",
           undef, $val);
```

This kind of test may be suitable for purposes such as checking a value submitted in a web form, but is inefficient for validating large datasets. It has no memory for the results of previous tests for values that have been seen before; consequently, you execute a query for every input value.

Construct a hash from the entire lookup table

To validate a large number of values, it's more efficient to pull the lookup values into memory, save them in a data structure, and check each input value against the contents of that structure. Using an in-memory lookup avoids the overhead of executing a query for each value.

First, run a query to retrieve all the lookup table values and construct a hash from them:

```
my %members;  # hash for lookup values
my $sth = $dbh->prepare ("SELECT val FROM $tbl_name");
$sth->execute ();
while (my ($val) = $sth->fetchrow_array ())
{
  $members{$val} = 1;
}
```

Perform a hash key existence test to check a given value:

```
$valid = exists ($members{$val});
```

This technique reduces database traffic to a single query. However, for a large lookup table, that could still be a lot of traffic, and you might not want to hold the entire table in memory.

Performing Lookups with Other Languages

The lookup example shown here uses a Perl hash to determine whether a given value is present in a set of values:

```
$valid = exists ($members{$val});
```

Similar data structures exist for other languages. In Ruby, use a hash, and check input values using the `has_key?` method:

```
valid = members.has_key?(val)
```

In PHP, use an associative array, and perform a key lookup with `isset()`:

```
$valid = isset ($members[$val]);
```

In Python, use a dictionary, and check input values using the `has_key()` method:

```
valid = members.has_key(val)
```

For lookups in Java, use a `HashMap`, and test values with the `containsKey()` method:

```
valid = members.containsKey (val);
```

The *transfer* directory of the `recipes` distribution contains some sample code for lookup operations in each language.

Remember already seen values to avoid database lookups

Another lookup technique mixes individual statements with a hash that stores lookup value existence information. This approach can be useful if you have a very large lookup table. Begin with an empty hash:

```
my %members;  # hash for lookup values
```

Then, for each value to be tested, check whether it's present in the hash. If not, execute a query to check whether the value is present in the lookup table, and record the result of the query in the hash. The validity of the input value is determined by the value associated with the key, not by the existence of the key:

```
if (!exists ($members{$val})) # haven't seen this value yet
{
  my $count = $dbh->selectrow_array (
                "SELECT COUNT(*) FROM $tbl_name WHERE val = ?",
                undef, $val);
  # store true/false to indicate whether value was found
  $members{$val} = ($count > 0);
```

```
    }
    $valid = $members{$val};
```

For this method, the hash acts as a cache, so that you execute a lookup query for any given value only once, no matter how many times it occurs in the input. For datasets that have repeated values, this approach avoids issuing a separate query for every single test, while requiring an entry in the hash only for each unique value. It thus stands between the other two approaches in terms of the trade-off between database traffic and program memory requirements for the hash.

Note that the hash is used in a different manner for this method than for the previous method. Previously, the existence of the input value as a key in the hash determined the validity of the value, and the value associated with the hash key was irrelevant. For the hash-as-cache method, the meaning of key existence in the hash changes from "it's valid" to "it's been tested before." For each key, the value associated with it indicates whether the input value is present in the lookup table. (If you store as keys only those values that are found to be in the lookup table, you issue a query for each instance of an invalid value in the input dataset, which is inefficient.)

12.10. Converting Two-Digit Year Values to Four-Digit Form

Problem

You must convert years in date values from two digits to four digits.

Solution

Let MySQL do this for you, or perform the operation yourself if MySQL's conversion rules aren't appropriate.

Discussion

Two-digit year values are a problem because the century is not explicit in the data values. If you know the range of years spanned by your input, you can add the century without ambiguity. Otherwise, you can only guess. For example, the date 10/2/69 would be interpreted by most people in the US as as October 2, 1969. But if it represents Mahatma Gandhi's birth date, the year is actually 1869.

One way to convert years to four digits is to let MySQL do it. If you store a date containing a two-digit year, MySQL automatically converts it to four-digit form. MySQL uses a transition point of 1970; it interprets values from 00 to 69 as the years 2000 to 2069, and values from 70 to 99 as the years 1970 to 1999. These rules are appropriate for year

values in the range from 1970 to 2069. If your values lie outside this range, add the proper century yourself before storing them into MySQL.

To use a different transition point, convert years to four-digit form yourself. Here's a general-purpose routine that converts two-digit years to four digits and supports an arbitrary transition point:

```
sub yy_to_ccyy
{
my ($year, $transition_point) = @_;

  $transition_point = 70 unless defined ($transition_point);
  $year += ($year >= $transition_point ? 1900 : 2000) if $year < 100;
  return $year;
}
```

The function uses MySQL's transition point (70) by default. An optional second argument may be given to provide a different transition point. `yy_to_ccyy()` also verifies that the year actually is less than 100 and needs converting before modifying it. That way you can pass year values regardless of whether they include the century. Some sample invocations using the default transition point have the following results:

```
$val = yy_to_ccyy (60);         # returns 2060
$val = yy_to_ccyy (1960);       # returns 1960 (no conversion done)
```

Suppose that you want to convert year values as follows, using a transition point of 50:

```
00 .. 49 -> 2000 .. 2049
50 .. 99 -> 1950 .. 1999
```

To do this, pass an explicit transition point argument to `yy_to_ccyy()`:

```
$val = yy_to_ccyy (60, 50);     # returns 1960
$val = yy_to_ccyy (1960, 50);   # returns 1960 (no conversion done)
```

The `yy_to_ccyy()` function is included in the *Cookbook_Utils.pm* library file.

12.11. Performing Validity Checking on Date or Time Subparts

Problem

A string passes a pattern test as a date or time, but you want to perform further validity checking.

Solution

Break the value into parts and perform the appropriate range checking on each part.

Discussion

Pattern matching may not be sufficient for date or time checking. For example, a value like `1947-15-19` might match a date pattern, but it's not a legal date. To perform more rigorous value testing, combine pattern matching with range checking. Break out the year, month, and day values, then check whether each is within the proper range. Years should be less than 9999 (MySQL represents dates to an upper limit of `9999-12-31`), month values must be in the range from 1 to 12, and days must be in the range from 1 to the number of days in the month. That last part is the trickiest: it's month-dependent, and also year-dependent for February because it changes for leap years.

Suppose that you're checking input dates in ISO format. In Recipe 12.6, we used the `is_iso_date()` function from the *Cookbook_Utils.pm* library file to perform a pattern match on a date string and break it into component values. `is_iso_date()` returns `undef` if the value doesn't satisfy a pattern that matches ISO date format. Otherwise, it returns a reference to an array containing the year, month, and day values. The *Cookbook_Utils.pm* file also contains `is_mmddyy_date()` and `is_ddmmyy_date()` routines that match dates in US or British format and return `undef` or a reference to an array of date parts. (The parts returned are always in year, month, day order, not the order in which the parts appear in the input date string.)

To perform additional checking on the result returned by any of those routines (assuming that the result is not `undef`), pass the date parts to `is_valid_date()`, another library function:

```
$valid = is_valid_date ($ref->[0], $ref->[1], $ref->[2]);
```

Or, more concisely:

```
$valid = is_valid_date (@{$ref});
```

`is_valid_date()` returns nonzero if the date is valid, 0 otherwise. It checks the parts of a date like this:

```
sub is_valid_date
{
my ($year, $month, $day) = @_;

  # year must be nonnegative, month and day must be positive
  return 0 if $year < 0 || $month < 1 || $day < 1;
  # check maximum limits on individual parts
  return 0 if $year > 9999
              || $month > 12
              || $day > days_in_month ($year, $month);
  return 1;
}
```

`is_valid_date()` requires separate year, month, and day values, not a date string. This requires that you break candidate values into components before invoking it, but makes

it applicable in more contexts. For example, you can use it to check dates like `12 February 2003` by mapping the month to its numeric value before calling `is_valid_date()`. If `is_valid_date()` took a string argument assumed to be in a specific date format, it would be much less general.

`is_valid_date()` uses a subsidiary function `days_in_month()` to determine the number of days in the month represented by the date. `days_in_month()` requires both the year and the month as arguments because if the month is 2 (February), the number of days depends on whether the year is a leap year. This means you *must* pass a four-digit year value: as discussed in Recipe 6.18, two-digit years are ambiguous with respect to the century, which makes proper leap-year testing impossible. The `days_in_month()` and `is_leap_year()` functions are based on techniques taken from that recipe:

```
sub is_leap_year
{
my $year = $_[0];

  return ($year % 4 == 0) && ((($year % 100) != 0) || ($year % 400) == 0);
}

sub days_in_month
{
my ($year, $month) = @_;
my @day_tbl = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
my $days = $day_tbl[$month-1];

  # add a day for Feb of leap years
  $days++ if $month == 2 && is_leap_year ($year);
  return $days;
}
```

To perform validity checking on time values, a similar procedure applies: verify that the value matches a time pattern and break it into components, then perform range-testing on the components. For times, the ranges are 0 to 23 for the hour, and 0 to 59 for the minute and second. Here is a function `is_24hr_time()` that checks for values in 24-hour format and returns the components:

```
sub is_24hr_time
{
my $s = $_[0];

  return undef unless $s =~ /^(\d{1,2})\D(\d{2})\D(\d{2})$/;
  return [ $1, $2, $3 ];  # return hour, minute, second
}
```

The following `is_ampm_time()` function is similar but looks for times in 12-hour format with an optional AM or PM suffix, converting PM times to 24-hour values:

```
sub is_ampm_time
{
my $s = $_[0];
```

```
    return undef unless $s =~ /^(\d{1,2})\D(\d{2})\D(\d{2})(?:\s*(AM|PM))?$/i;
    my ($hour, $min, $sec) = ($1, $2, $3);
    if ($hour == 12 && (!defined ($4) || uc ($4) eq "AM"))
    {
      $hour = "00"; # 12:xx:xx AM times are 00:xx:xx
    }
    elsif ($hour < 12 && defined ($4) && uc ($4) eq "PM")
    {
      $hour += 12;  # PM times other than 12:xx:xx
    }
    return [ $hour, $min, $sec ]; # return hour, minute, second
}
```

Both functions return `undef` for values that don't match the pattern. Otherwise, they return a reference to a three-element array containing the hour, minute, and second values.

After you obtain the time components, pass them to `is_valid_time()`, another utility routine, to perform range checks.

12.12. Writing Date-Processing Utilities

Problem

There's a date-processing operation that you must perform frequently. You want to write a utility that does it for you.

Solution

The utilities in this recipe provide some examples that show how to do that.

Discussion

Due to the idiosyncratic nature of dates, you might occasionally find it necessary to write date converters. This section shows some sample converters that serve various purposes:

- *isoize_date.pl* reads a file looking for dates in US format (*MM-DD-YY*) and converts them to ISO format.
- *cvt_date.pl* converts dates to and from any of ISO, US, or British formats. It is more general than *isoize_date.pl*, but requires that you tell it what kind of input to expect and what kind of output to produce.

- *monddccyy_to_iso.pl* looks for dates like `Feb. 6, 1788` and converts them to ISO format. It illustrates how to map dates with nonnumeric parts to a format that MySQL understands.

All three scripts are located in the *transfer* directory of the `recipes` distribution. They assume datafiles are in tab-delimited, linefeed-terminated format. To work with files that have a different format, use *cvt_file.pl* (see Recipe 11.6).

Our first date-processing utility, *isoize_date.pl*, looks for dates in US format and rewrites them into ISO format. You'll recognize that it's modeled after the general input-processing loop shown in Recipe 12.2, with some extra stuff thrown in to perform a specific type of conversion:

```
#!/usr/bin/perl
# isoize_date.pl: Read input data, look for values that match
# a date pattern, convert them to ISO format. Also converts
# 2-digit years to 4-digit years, using a transition point of 70.
# By default, this looks for dates in MM-DD-[CC]YY format.
# Does not check whether dates actually are valid (for example,
# won't complain about 13-49-1928).

# Assumes tab-delimited, linefeed-terminated input lines.

use strict;
use warnings;

# transition point at which 2-digit XX year values are assumed to be
# 19XX (below that, they are treated as 20XX)
my $transition = 70;

while (<>)
{
  chomp;
  my @val = split (/\t/, $_, 10000);  # split, preserving all fields
  for my $i (0 .. @val - 1)
  {
    # look for strings in MM-DD-[CC]YY format
    next unless $val[$i] =~ /^(\d{1,2})\D(\d{1,2})\D(\d{2,4})$/;

    my ($month, $day, $year) = ($1, $2, $3);
    # to interpret dates as DD-MM-[CC]YY instead, replace preceding
    # line with the following one:
    #my ($day, $month, $year) = ($1, $2, $3);

    # convert 2-digit years to 4 digits, then update value in array
    $year += ($year >= $transition ? 1900 : 2000) if $year < 100;
    $val[$i] = sprintf ("%04d-%02d-%02d", $year, $month, $day);
  }
  print join ("\t", @val) . "\n";
}
```

If you feed *isoize_date.pl* an input file that looks like this:

```
Sybil   04-13-70
Nancy   09-30-69
Ralph   11-02-73
Lothair 07-04-63
Henry   02-14-65
Aaron   09-17-68
Joanna  08-20-52
Stephen 05-01-60
```

It produces the following output:

```
Sybil   1970-04-13
Nancy   2069-09-30
Ralph   1973-11-02
Lothair 2063-07-04
Henry   2065-02-14
Aaron   2068-09-17
Joanna  2052-08-20
Stephen 2060-05-01
```

isoize_date.pl serves a specific purpose: it converts only from US to ISO format. It does not perform validity checking on date subparts or permit the transition point for adding the century to be specified. A more general tool would be more useful. The next script, *cvt_date.pl*, extends the capabilities of *isoize_date.pl*; it recognizes input dates in ISO, US, or British formats and converts any of them to any other. It also can convert two-digit years to four digits, enable you to specify the conversion transition point, and warn about bad dates. As such, it can be used to preprocess input for loading into MySQL or postprocess data exported from MySQL for use by other programs.

cvt_date.pl understands the following options:

`--iformat=`*`format`*, `--oformat=`*`format`*, `--format=`*`format`*
: Set the date format for input, output, or both. The default *`format`* value is `iso`; *cvt_date.pl* also recognizes any string beginning with `us` or `br` as indicating US or British date format.

`--add-century`
: Convert two-digit years to four digits.

`--columns=`*`column_list`*
: Convert dates only in the named columns. By default, *cvt_date.pl* looks for dates in all columns. If this option is given, *`column_list`* should be a list of one or more column positions or ranges separated by commas. (Ranges can be given as *`m-n`* to specify columns *`m`* through *`n`*.) Positions begin at 1.

`--transition=`*`n`*
: Specify the transition point for two-digit to four-digit year conversions. The default transition point is 70. This option turns on `--add-century`.

`--warn`
: Warn about bad dates. (This option can produce spurious warnings if the dates have two-digit years and you don't specify `--add-century`, because leap-year testing won't always be accurate in that case.)

I won't show the code for *cvt_date.pl* here (most of it is taken up with processing command-line options), but you can examine the source for yourself if you like. As an example of how *cvt_date.pl* works, suppose that you have a file *newdata.txt* with the following contents:

```
name1   01/01/99    38
name2   12/31/00    40
name3   02/28/13    42
name4   01/02/18    44
```

Running the file through *cvt_date.pl* with options indicating that the dates are in US format and that the century should be added produces this result:

```
% cvt_date.pl --iformat=us --add-century newdata.txt
name1   1999-01-01  38
name2   2000-12-31  40
name3   2013-02-28  42
name4   2018-01-02  44
```

To produce dates in British format instead with no year conversion, do this:

```
% cvt_date.pl --iformat=us --oformat=br newdata.txt
name1   01-01-99    38
name2   31-12-00    40
name3   28-02-13    42
name4   02-01-18    44
```

cvt_date.pl has no knowledge of the meaning of each data column, of course. If you have a nondate column with values that match the pattern, it rewrites that column, too. To deal with that, specify a `--columns` option to limit the columns that *cvt_date.pl* converts.

isoize_date.pl and *cvt_date.pl* both operate on dates written in all-numeric formats. But dates in datafiles often are written differently, and it may be necessary to write a special-purpose script to process them. Suppose an input file contains dates in the following format (these represent the dates on which US states were admitted to the Union):

```
Delaware        Dec. 7, 1787
Pennsylvania    Dec 12, 1787
New Jersey      Dec. 18, 1787
Georgia         Jan. 2, 1788
Connecticut     Jan. 9, 1788
Massachusetts   Feb. 6, 1788
…
```

The dates consist of a three-character month abbreviation (possibly followed by a period), a numeric day of the month, a comma, and a numeric year. To import this file

into MySQL, you must convert the dates to ISO format, resulting in a file that looks like this:

```
Delaware        1787-12-07
Pennsylvania    1787-12-12
New Jersey      1787-12-18
Georgia         1788-01-02
Connecticut     1788-01-09
Massachusetts   1788-02-06
…
```

That's a somewhat specialized kind of transformation, although this general type of problem (converting a particular date format to ISO format) is hardly uncommon. To perform the conversion, identify the dates as those values matching an appropriate pattern, map month names to the corresponding numeric values, and reformat the result. The following script, *monddccyy_to_iso.pl*, illustrates how:

```
#!/usr/bin/perl
# monddccyy_to_iso.pl: Convert dates from mon[.] dd, ccyy to ISO format.

# Assumes tab-delimited, linefeed-terminated input

use strict;
use warnings;

my %map =   # map 3-char month abbreviations to numeric month
(
  "jan" => 1, "feb" => 2, "mar" => 3, "apr" => 4, "may" => 5, "jun" => 6,
  "jul" => 7, "aug" => 8, "sep" => 9, "oct" => 10, "nov" => 11, "dec" => 12
);

while (<>)
{
  chomp;
  my @val = split (/\t/, $_, 10000);    # split, preserving all fields
  for my $i (0 .. @val - 1)
  {
    # reformat the value if it matches the pattern, otherwise assume
    # that it's not a date in the required format and leave it alone
    if ($val[$i] =~ /^([^.]+)\.? (\d+), (\d+)$/)
    {
      # use lowercase month name
      my ($month, $day, $year) = (lc ($1), $2, $3);
      if (exists ($map{$month}))
      {
        $val[$i] = sprintf ("%04d-%02d-%02d", $year, $map{$month}, $day);
      }
      else
      {
        # warn, but don't reformat
        warn "$val[$i]: bad date?\n";
      }
```

```
    }
  }
  print join ("\t", @val) . "\n";
}
```

The script only does reformatting, it doesn't validate the dates. To do that, modify the script to use the *Cookbook_Utils.pm* module by adding this statement after the `use warnings` line:

```
use Cookbook_Utils;
```

That gives the script access to the module's `is_valid_date()` routine. To use it, change this line:

```
if (exists ($map{$month}))
```

To this:

```
if (exists ($map{$month}) && is_valid_date ($year, $map{$month}, $day))
```

12.13. Importing Non-ISO Date Values

Problem

Date values to be imported are not in the ISO (*CCYY-MM-DD*) format that MySQL expects.

Solution

Use an external utility to convert the dates to ISO format before importing the data into MySQL (*cvt_date.pl* is useful here). Or use `LOAD DATA`'s capability for preprocessing input data prior to loading it into the database.

Discussion

Suppose that a table contains three columns, `name`, `date`, and `value`, where `date` is a `DATE` column requiring values in ISO format (*CCYY-MM-DD*). Suppose also that you're given a datafile *newdata.txt* to be imported into the table, but its contents look like this:

```
name1   01/01/99    38
name2   12/31/00    40
name3   02/28/13    42
name4   01/02/18    44
```

The dates are in *MM/DD/YY* format and must be converted to ISO format to be stored as `DATE` values in MySQL. One way to do this is to run the file through the *cvt_date.pl* script from Recipe 12.12:

```
% cvt_date.pl --iformat=us --add-century newdata.txt > tmp.txt
```

Then load the *tmp.txt* file into the table. This task also can be accomplished entirely in MySQL with no external utilities by using SQL to perform the reformatting operation. As discussed in Recipe 11.1, `LOAD DATA` can preprocess input values before inserting them. Applying that capability to the present problem, the date-rewriting `LOAD DATA` statement looks like this, using the `STR_TO_DATE()` function (see Recipe 6.3) to interpret the input dates:

```
mysql> LOAD DATA LOCAL INFILE 'newdata.txt'
    -> INTO TABLE t (name,@date,value)
    -> SET date = STR_TO_DATE(@date,'%m/%d/%y');
```

With the `%y` format specifier in `STR_TO_DATE()`, MySQL converts the two-digit years to four-digit years automatically, so the original *MM/DD/YY* values end up as ISO values in *CCYY-MM-DD* format. The resulting data after import looks like this:

```
+-------+------------+-------+
| name  | date       | value |
+-------+------------+-------+
| name1 | 1999-01-01 |    38 |
| name2 | 2000-12-31 |    40 |
| name3 | 2013-02-28 |    42 |
| name4 | 2018-01-02 |    44 |
+-------+------------+-------+
```

This procedure assumes that MySQL's automatic conversion of two-digit years to four digits produces the correct century values. This means that the year part of the values must correspond to years in the range from 1970 to 2069. If that's not true, you must convert the year values some other way. (For some ideas, see Recipe 12.11.)

If the dates are not in a format that `STR_TO_DATE()` can interpret, perhaps you can write a stored function to handle them and return ISO date values. In that case, the `LOAD DATA` statement looks like this, where `my_date_interp()` is your stored function name:

```
mysql> LOAD DATA LOCAL INFILE 'newdata.txt'
    -> INTO TABLE t (name,@date,value)
    -> SET date = my_date_interp(@date);
```

12.14. Exporting Dates Using Non-ISO Formats

Problem

You want to export date values using a format other than MySQL's default ISO (*CCYY-MM-DD*) format. This might be a requirement when exporting dates from MySQL to applications that don't use ISO format.

Solution

Use an external utility to rewrite the dates to non-ISO format after exporting the data from MySQL (*cvt_date.pl* is useful here). Or use the `DATE_FORMAT()` function to rewrite the values during the export operation.

Discussion

Suppose that you want to export data from MySQL into an application that doesn't understand ISO-format dates. One way to do this is to export the data into a file, leaving the dates in ISO format. Then run the file through a utility such as *cvt_date.pl* that rewrites the dates into the required format (see Recipe 12.12).

Another approach is to export the dates directly in the required format by rewriting them with `DATE_FORMAT()`. Suppose that you have the following table:

```
CREATE TABLE datetbl
(
  i   INT,
  c   CHAR(10),
  d   DATE,
  dt  DATETIME,
  ts  TIMESTAMP
);
```

Suppose also that you need to export data from this table, but with the dates in any `DATE`, `DATETIME`, or `TIMESTAMP` columns rewritten in US format (*`MM-DD-CCYY`*). A `SELECT` statement that uses the `DATE_FORMAT()` function to rewrite the dates as required looks like this:

```
SELECT
  i,
  c,
  DATE_FORMAT(d, '%m-%d-%Y') AS d,
  DATE_FORMAT(dt, '%m-%d-%Y %T') AS dt,
  DATE_FORMAT(ts, '%m-%d-%Y %T') AS ts
FROM datetbl
```

If `datetbl` contains the following rows:

```
3       abc     2005-12-31      2005-12-31 12:05:03     2005-12-31 12:05:03
4       xyz     2006-01-31      2006-01-31 12:05:03     2006-01-31 12:05:03
```

The statement generates output that looks like this:

```
3       abc     12-31-2005      12-31-2005 12:05:03     12-31-2005 12:05:03
4       xyz     01-31-2006      01-31-2006 12:05:03     01-31-2006 12:05:03
```

12.15. Epilogue

Recall the scenario presented at the beginning of Chapter 11:

Suppose that a file named *somedata.csv* contains 12 data columns in comma-separated values (CSV) format. From this file you want to extract only columns 2, 11, 5, and 9, and use them to create database rows in a MySQL table that contains `name`, `birth`, `height`, and `weight` columns. You must make sure that the height and weight are positive integers, and convert the birth dates from *MM/DD/YY* format to *CCYY-MM-DD* format. How can you do this?

So... how *can* you do that, based on the techniques discussed in the previous chapter and this one?

Much of the work can be done using the utility programs developed here. Convert the file to tab-delimited format with *cvt_file.pl* (see Recipe 11.6), extract the columns in the desired order with *yank_col.pl* (see Recipe 11.7), and rewrite the date column to ISO format with *cvt_date.pl* (see Recipe 12.12):

```
% cvt_file.pl --iformat=csv somedata.csv \
    | yank_col.pl --columns=2,11,5,9 \
    | cvt_date.pl --columns=2 --iformat=us --add-century > tmp
```

The resulting file, *tmp*, has four columns representing the `name`, `birth`, `height`, and `weight` values, in that order. It needs only to have its height and weight columns checked to make sure they contain positive integers. Using the `is_positive_integer()` library function from the *Cookbook_Utils.pm* module file, that task can be achieved using a short special-purpose script that is little more than an input loop:

```
#!/usr/bin/perl
# validate_htwt.pl: Height/weight validation example.

# Assumes tab-delimited, linefeed-terminated input lines.

# Input columns and the actions to perform on them are as follows:
# 1: name; echo as given
# 2: birth; echo as given
# 3: height; validate as positive integer
# 4: weight; validate as positive integer

use strict;
use warnings;
use Cookbook_Utils;

while (<>)
{
  chomp;
  my ($name, $birth, $height, $weight) = split (/\t/, $_, 4);
  warn "line $.:height $height is not a positive integer\n"
               if !is_positive_integer ($height);
  warn "line $.:weight $weight is not a positive integer\n"
               if !is_positive_integer ($weight);
}
```

The *validate_htwt.pl* script produces no output (except for warning messages) because it need not reformat any of the input values. If *tmp* passes validation with no errors, it can be loaded into MySQL with a simple `LOAD DATA` statement:

```
mysql> LOAD DATA LOCAL INFILE 'tmp' INTO TABLE tbl_name;
```

CHAPTER 13

Generating and Using Sequences

13.0. Introduction

A sequence is a set of integers (1, 2, 3, …) generated in order on demand. Sequences see frequent use in databases because many applications require each row in a table to contain a unique value, and sequences provide an easy way to generate them. This chapter describes how to use sequences in MySQL:

Using `AUTO_INCREMENT` *columns*
: The `AUTO_INCREMENT` column is MySQL's mechanism for generating a sequence over a set of rows. Each time you create a row in a table that contains an `AUTO_INCREMENT` column, MySQL automatically generates the next value in the sequence as the column's value. This value serves as a unique identifier, making sequences an easy way to create items such as customer ID numbers, shipping package waybill numbers, invoice or purchase order numbers, bug report IDs, ticket numbers, or product serial numbers.

Retrieving sequence values
: For many applications, it's not enough just to create sequence values. It's also necessary to determine the sequence value for a just-inserted row. A web application may need to redisplay to a user the contents of a row created from the contents of a form just submitted by the user. The value may need to be retrieved so it can be stored in rows of a related table.

Resequencing techniques
: It's possible to renumber a sequence that has holes in it due to row deletions, reuse deleted values at the top of a sequence, or add a sequence column to a table that has none.

Managing multiple simultaneous sequences
: Special care is necessary when you need to keep track of multiple sequence values, such as when you create rows in multiple tables that each have an `AUTO_INCREMENT` column.

Using single-row sequence generators
: Sequences can be used as counters. For example, to count votes in a poll, you might increment a counter each time a candidate receives a vote. The counts for a given candidate form a sequence, but because the count itself is the only value of interest, there is no need to generate a new row to record each vote. MySQL provides a solution for this problem using a mechanism that enables a sequence to be easily generated within a single table row over time. To store multiple counters in the table, use a column that identifies each counter uniquely. The same mechanism also enables creation of sequences that increase by values other than one or by nonuniform values.

The engines for most database systems provide sequence-generation capabilities, although the implementations tend to be engine-dependent. That's true for MySQL as well, so the material in this section is almost completely MySQL-specific, even at the SQL level. In other words, the SQL for generating sequences is itself nonportable, even if you use an API such as DBI or JDBC that provides an abstraction layer. Abstract interfaces may help you process SQL statements portably, but they don't make nonportable SQL portable.

Scripts related to the examples shown in this chapter are located in the *sequences* directory of the `recipes` distribution. For scripts that create tables used here, look in the *tables* directory.

13.1. Creating a Sequence Column and Generating Sequence Values

Problem

A table must include a column containing unique IDs.

Solution

Use an `AUTO_INCREMENT` column to generate a sequence.

Discussion

This section provides the essential background on using `AUTO_INCREMENT` columns, beginning with an example that demonstrates the sequence-generation mechanism. The illustration centers around a bug-collection scenario: your eight-year-old son Junior is

assigned the task of collecting insects for a class project at school. For each insect, Junior is to record its name ("ant," "bee," and so forth), and its date and location of collection. You have expounded the benefits of MySQL for record-keeping to Junior since his early days, so upon your arrival home from work that day, he immediately announces the necessity of completing this project and then, looking you straight in the eye, declares that it's clearly a task for which MySQL is well-suited. Who are you to argue? So the two of you get to work. Junior already collected some specimens after school while waiting for you to come home and has recorded the following information in his notebook:

Name	Date	Origin
millipede	2014-09-10	driveway
housefly	2014-09-10	kitchen
grasshopper	2014-09-10	front yard
stink bug	2014-09-10	front yard
cabbage butterfly	2014-09-10	garden
ant	2014-09-10	back yard
ant	2014-09-10	back yard
termite	2014-09-10	kitchen woodwork

Looking over Junior's notes, you're pleased to see that even at his tender age, he has learned to write dates in ISO format. However, you also notice that he's collected a millipede and a termite, neither of which actually are insects. You decide to let this pass for the moment; Junior forgot to bring home the written instructions for the project, so at this point it's unclear whether these specimens are acceptable. (You also note with some alarm Junior's discovery of termites in the house and make a mental note to call the exterminator.)

As you consider how to create a table to store this information, it's apparent that you need at least `name`, `date`, and `origin` columns corresponding to the types of information that Junior is required to record:

```
CREATE TABLE insect
(
  name    VARCHAR(30) NOT NULL,   # type of insect
  date    DATE NOT NULL,          # date collected
  origin  VARCHAR(30) NOT NULL    # where collected
);
```

However, those columns are insufficient to make the table easy to use. Note that the records collected thus far are not unique; both ants were collected at the same time and place. If you put the information into an `insect` table that has the structure just shown, neither ant row can be referred to individually because there's nothing to distinguish one from another. Unique IDs would be helpful to make the rows distinct and to provide

values that make each row easy to refer to. An AUTO_INCREMENT column is good for this purpose, so a better insect table has a structure like this:

```
CREATE TABLE insect
(
  id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (id),
  name    VARCHAR(30) NOT NULL,   # type of insect
  date    DATE NOT NULL,          # date collected
  origin  VARCHAR(30) NOT NULL    # where collected
);
```

Go ahead and create the insect table using this second CREATE TABLE statement. (Recipe 13.2 discusses the particulars of the id column definition.)

Now that you have an AUTO_INCREMENT column, use it to generate new sequence values. One of the useful properties of an AUTO_INCREMENT column is that you need not assign its values yourself: MySQL does so for you. There are two ways to generate new AUTO_INCREMENT values in the id column. One is to explicitly set the id column to NULL. The following statement inserts the first four of Junior's specimens into the insect table that way:

```
mysql> INSERT INTO insect (id,name,date,origin) VALUES
    -> (NULL,'housefly','2014-09-10','kitchen'),
    -> (NULL,'millipede','2014-09-10','driveway'),
    -> (NULL,'grasshopper','2014-09-10','front yard'),
    -> (NULL,'stink bug','2014-09-10','front yard');
```

Alternatively, omit the id column from the INSERT statement entirely. MySQL permits creating rows without explicitly specifying values for columns that have a default value. MySQL assigns each missing column its default value, and the default for an AUTO_INCREMENT column is its next sequence number. Thus, this statement adds Junior's other four specimens to the insect table and generates sequence values without naming the id column at all:

```
mysql> INSERT INTO insect (name,date,origin) VALUES
    -> ('cabbage butterfly','2014-09-10','garden'),
    -> ('ant','2014-09-10','back yard'),
    -> ('ant','2014-09-10','back yard'),
    -> ('termite','2014-09-10','kitchen woodwork');
```

Whichever method you use, MySQL determines the sequence number for each row and assigns it to the id column, as you can verify:

```
mysql> SELECT * FROM insect ORDER BY id;
+----+-------------------+------------+-----------------+
| id | name              | date       | origin          |
+----+-------------------+------------+-----------------+
|  1 | housefly          | 2014-09-10 | kitchen         |
|  2 | millipede         | 2014-09-10 | driveway        |
|  3 | grasshopper       | 2014-09-10 | front yard      |
```

```
|  4 | stink bug         | 2014-09-10 | front yard       |
|  5 | cabbage butterfly | 2014-09-10 | garden           |
|  6 | ant               | 2014-09-10 | back yard        |
|  7 | ant               | 2014-09-10 | back yard        |
|  8 | termite           | 2014-09-10 | kitchen woodwork |
+----+-------------------+------------+------------------+
```

As Junior collects more specimens, add more rows to the table and they'll be assigned the next values in the sequence (9, 10, …).

The concept underlying `AUTO_INCREMENT` columns is simple enough in principle: each time you create a new row, MySQL generates the next number in the sequence and assigns it to the row. But there are certain subtleties to know about, as well as differences in how different storage engines handle `AUTO_INCREMENT` sequences. Awareness of these issues enables you to use sequences more effectively and avoid surprises. For example, if you explicitly set the `id` column to a non-`NULL` value, one of two things happens:

- If the value is already present in the table, an error occurs if the column cannot contain duplicates. For the `insect` table, the `id` column is a `PRIMARY KEY`, which prohibits duplicates:

```
mysql> INSERT INTO insect (id,name,date,origin) VALUES
    -> (3,'cricket','2014-09-11','basement');
ERROR 1062 (23000): Duplicate entry '3' for key 'PRIMARY'
```

- If the value is not present in the table, MySQL inserts the row using that value. In addition, if the value is larger than the current sequence counter, the table's counter is reset to the value plus one. The `insect` table at this point has sequence values 1 through 8. If you insert a new row with the `id` column set to 20, that becomes the new maximum value. Subsequent inserts that automatically generate `id` values will begin at 21. The values 9 through 19 become unused, resulting in a gap in the sequence.

The next recipe looks in more detail at how to define `AUTO_INCREMENT` columns and how they behave.

13.2. Choosing the Definition for a Sequence Column

Problem

You want to know more about how to define a sequence column.

Solution

Use the guidelines given here.

Discussion

You should follow certain principles when creating AUTO_INCREMENT columns. As an illustration, consider how Recipe 13.1 declared the id column in the insect table:

```
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
PRIMARY KEY (id)
```

The AUTO_INCREMENT keyword informs MySQL that it should generate successive sequence numbers for the column's values, but the other information is important, too:

- INT is the column's base data type. You need not necessarily use INT, but the column should be one of the integer types: TINYINT, SMALLINT, MEDIUMINT, INT, or BIGINT.
- UNSIGNED prohibits negative column values. This is not a required attribute for AUTO_INCREMENT columns, but sequences consist only of positive integers (normally beginning at 1), so there is no reason to permit negative values. Furthermore, *not* declaring the column to be UNSIGNED cuts the range of your sequence in half. For example, TINYINT has a range of –128 to 127. Because sequences include only positive values, the effective range of a TINYINT sequence is 1 to 127. TINYINT UNSIGNED has a range of 0 to 255, which increases the upper end of the sequence to 255. The specific integer type determines the maximum sequence value. The following table shows the maximum unsigned value of each type; use this information to choose a type big enough to hold the largest value you'll need:

Data type	Maximum unsigned value
TINYINT	255
SMALLINT	65,535
MEDIUMINT	16,777,215
INT	4,294,967,295
BIGINT	18,446,744,073,709,551,615

  Sometimes people omit UNSIGNED so that they can create rows that contain negative numbers in the sequence column (using –1 to signify "has no ID," for example.) This is a bad idea. MySQL makes no guarantees about how negative numbers will be treated in an AUTO_INCREMENT column, so by using them you're playing with fire. For example, if you resequence the column, all your negative values get turned into positive sequence numbers.
- AUTO_INCREMENT columns cannot contain NULL values, so id is declared as NOT NULL. (It's true that you can specify NULL as the column value when you insert a new row, but for an AUTO_INCREMENT column, that really means "generate the next sequence value.") MySQL automatically defines AUTO_INCREMENT columns as NOT NULL if you forget.

- AUTO_INCREMENT columns must be indexed. Normally, because a sequence column exists to provide unique identifiers, you use a PRIMARY KEY or UNIQUE index to enforce uniqueness. Tables can have only one PRIMARY KEY, so if the table already has some other PRIMARY KEY column, you can declare an AUTO_INCREMENT column to have a UNIQUE index instead:

```
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
UNIQUE (id)
```

When you create a table that contains an AUTO_INCREMENT column, it's also important to consider which storage engine to use (InnoDB, MyISAM, and so forth). The engine affects behaviors such as reuse of values deleted from the top of the sequence (see Recipe 13.3).

13.3. The Effect of Row Deletions on Sequence Generation

Problem

You want to know what happens to a sequence when you delete rows from a table that contains an AUTO_INCREMENT column.

Solution

It depends on which rows you delete and on the storage engine.

Discussion

We have thus far considered how MySQL generates sequence values in an AUTO_INCREMENT column under circumstances where rows are only added to a table. But it's unrealistic to assume that rows will never be deleted. What happens to the sequence then?

Refer again to Junior's bug-collection project, for which you currently have an insect table that looks like this:

```
mysql> SELECT * FROM insect ORDER BY id;
+----+-------------------+------------+-----------------+
| id | name              | date       | origin          |
+----+-------------------+------------+-----------------+
|  1 | housefly          | 2014-09-10 | kitchen         |
|  2 | millipede         | 2014-09-10 | driveway        |
|  3 | grasshopper       | 2014-09-10 | front yard      |
|  4 | stink bug         | 2014-09-10 | front yard      |
|  5 | cabbage butterfly | 2014-09-10 | garden          |
|  6 | ant               | 2014-09-10 | back yard       |
|  7 | ant               | 2014-09-10 | back yard       |
|  8 | termite           | 2014-09-10 | kitchen woodwork |
+----+-------------------+------------+-----------------+
```

That's about to change because after Junior remembers to bring home the written instructions for the project, you read through them and discover two things that affect the table contents:

- Specimens should include only insects, not insect-like creatures such as millipedes and termites.
- The purpose of the project is to collect as many *different* specimens as possible, not just as *many* specimens as possible. This means that only one ant row is permitted.

These instructions dictate that a few rows be removed from table—specifically those with `id` values 2 (millipede), 8 (termite), and 7 (duplicate ant). Thus, despite Junior's evident disappointment at the reduction in the size of his collection, you instruct him to remove those rows by issuing a `DELETE` statement:

```
mysql> DELETE FROM insect WHERE id IN (2,8,7);
```

This statement illustrates why it's useful to have unique ID values: they enable you to specify any row unambiguously. The ant rows are identical except for the `id` value. Without that column in the table, it would be more difficult to delete just one of them (though not impossible; see Recipe 16.4).

After removing the unsuitable rows, the table has these remaining:

```
mysql> SELECT * FROM insect ORDER BY id;
+----+-------------------+------------+------------+
| id | name              | date       | origin     |
+----+-------------------+------------+------------+
|  1 | housefly          | 2014-09-10 | kitchen    |
|  3 | grasshopper       | 2014-09-10 | front yard |
|  4 | stink bug         | 2014-09-10 | front yard |
|  5 | cabbage butterfly | 2014-09-10 | garden     |
|  6 | ant               | 2014-09-10 | back yard  |
+----+-------------------+------------+------------+
```

The `id` column sequence now has a hole (row 2 is missing) and the values 7 and 8 at the top of the sequence are no longer present. How do these deletions affect future insert operations? What sequence number will the next new row get?

Removing row 2 creates a gap in the middle of the sequence. This has no effect on subsequent inserts, because MySQL makes no attempt to fill in holes in a sequence. On the other hand, deleting rows 7 and 8 removes values at the top of the sequence. For InnoDB or MyISAM tables, values are not reused. The next sequence number is the smallest positive integer that has not previously been used. (For a sequence that stands at 8, the next row gets a value of 9 even if you delete rows 7 and 8 first.) If you require strictly monotonic sequences, you can use one of these storage engines. For other storage engines, values removed at the top of the sequence may or may not be reused. Check the properties of the engine before using it.

If a table uses an engine that differs in value-reuse behavior from the behavior you require, use `ALTER TABLE` to change the table to a more appropriate engine. For example, to change a table to use InnoDB (to prevent sequence values from being reused after rows are deleted), do this:

```
ALTER TABLE tbl_name ENGINE = InnoDB;
```

If you don't know what engine a table uses, consult `INFORMATION_SCHEMA` or use `SHOW TABLE STATUS` or `SHOW CREATE TABLE` to find out. For example, the following statement indicates that `insect` is an InnoDB table:

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'insect';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+
```

To empty a table and reset the sequence counter (even for engines that normally do not reuse values), use `TRUNCATE TABLE`:

```
TRUNCATE TABLE tbl_name;
```

13.4. Retrieving Sequence Values

Problem

After creating a row that includes a new sequence number, you want to know what that number is.

Solution

Invoke the `LAST_INSERT_ID()` function. If you're writing a program, your MySQL API may provide a way to get the value directly without issuing an SQL statement.

Discussion

It's common for applications to need to know the `AUTO_INCREMENT` value of a newly created row. For example, if you write a web-based frontend for entering rows into Junior's `insect` table, you might have the application display each new row nicely formatted in a new page immediately after you hit the Submit button. To do this, you must know the new `id` value so that you can retrieve the proper row. Another situation in which the `AUTO_INCREMENT` value is needed occurs when you use multiple tables: after inserting a row in a master table, you need its ID to create rows in other related tables that refer to the master row. (Recipe 13.11 shows how to do this.)

When you generate a new AUTO_INCREMENT value, one way to get the value from the server is to execute a statement that invokes the LAST_INSERT_ID() function. In addition, many MySQL APIs provide a client-side mechanism for making the value available without issuing another statement. This recipe discusses both methods and compares their characteristics.

Using LAST_INSERT_ID() to obtain AUTO_INCREMENT values

The obvious (but incorrect) way to determine a new row's AUTO_INCREMENT value uses the fact that when MySQL generates the value, it becomes the largest sequence number in the column. Thus, you might try using the MAX() function to retrieve it:

```
SELECT MAX(id) FROM insect;
```

This is unreliable; if another client inserts a row before you issue the SELECT statement, MAX(id) returns that client's ID, not yours. It's possible to solve this problem by grouping the INSERT and SELECT statements as a transaction or locking the table, but MySQL provides a simpler way to obtain the proper value: invoke the LAST_INSERT_ID() function. It returns the most recent AUTO_INCREMENT value generated within your session, regardless of what other clients are doing. For example, to insert a row into the insect table and retrieve its id value, do this:

```
mysql> INSERT INTO insect (name,date,origin)
    -> VALUES('cricket','2014-09-11','basement');
mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                9 |
+------------------+
```

Or you can use the new value to retrieve the entire row, without even knowing what it is:

```
mysql> INSERT INTO insect (name,date,origin)
    -> VALUES('moth','2014-09-14','windowsill');
mysql> SELECT * FROM insect WHERE id = LAST_INSERT_ID();
+----+------+------------+------------+
| id | name | date       | origin     |
+----+------+------------+------------+
| 10 | moth | 2014-09-14 | windowsill |
+----+------+------------+------------+
```

The server maintains the value returned by LAST_INSERT_ID() on a session-specific basis. This property is by design, and it's important because it prevents clients from interfering with each other. When you generate an AUTO_INCREMENT value, LAST_INSERT_ID() returns that specific value, even when other clients generate new rows in the same table in the meantime.

Using API-specific methods to obtain AUTO_INCREMENT values

`LAST_INSERT_ID()` is an SQL function, so you can use it from within any client that can execute SQL statements. On the other hand, you do have to execute a separate statement to get its value. When you write your own programs, you may have another choice. Many MySQL interfaces include an API-specific extension that returns the `AUTO_INCREMENT` value without executing an additional statement. Most of our APIs have this capability.

Perl

Use the `mysql_insertid` attribute to obtain the `AUTO_INCREMENT` value generated by a statement. This attribute is accessed through either a database handle or a statement handle, depending on how you issue the statement. The following example references it through the database handle:

```
$dbh->do ("INSERT INTO insect (name,date,origin)
           VALUES('moth','2014-09-14','windowsill')");
my $seq = $dbh->{mysql_insertid};
```

To access `mysql_insertid` as a statement-handle attribute, use `prepare()` and `execute()`:

```
my $sth = $dbh->prepare ("INSERT INTO insect (name,date,origin)
                          VALUES('moth','2014-09-14','windowsill')");
$sth->execute ();
my $seq = $sth->{mysql_insertid};
```

Ruby

The Ruby DBI driver for MySQL exposes the client-side `AUTO_INCREMENT` value using the `func` database-handle method that returns driver-specific values:

```
dbh.do("INSERT INTO insect (name,date,origin)
        VALUES('moth','2014-09-14','windowsill')")
seq = dbh.func(:insert_id)
```

PHP

The PDO interface for MySQL has a `lastInsertId()` database-handle method that returns the most recent `AUTO_INCREMENT` value:

```
$dbh->exec ("INSERT INTO insect (name,date,origin)
             VALUES('moth','2014-09-14','windowsill')");
$seq = $dbh->lastInsertId ();
```

Python

The Connector/Python driver for DB API provides a `lastrowid` cursor object attribute that returns the most recent `AUTO_INCREMENT` value:

```
cursor = conn.cursor()
cursor.execute('''
               INSERT INTO insect (name,date,origin)
               VALUES('moth','2014-09-14','windowsill')
```

```
                ''')
seq = cursor.lastrowid
```

Java

The Connector/J JDBC driver `getGeneratedKeys()` method returns `AUTO_INCREMENT` values. It can be used with a `Statement` or `PreparedStatement` object if you supply an additional `Statement.RETURN_GENERATED_KEYS` argument during the statement-execution process to indicate that you want to retrieve the sequence value.

For a `Statement`:

```
Statement s = conn.createStatement ();
s.executeUpdate ("INSERT INTO insect (name,date,origin)"
                 + " VALUES('moth','2014-09-14','windowsill')",
                 Statement.RETURN_GENERATED_KEYS);
```

For a `PreparedStatement`:

```
PreparedStatement s = conn.prepareStatement (
                    "INSERT INTO insect (name,date,origin)"
                    + " VALUES('moth','2014-09-14','windowsill')",
                    Statement.RETURN_GENERATED_KEYS);
s.executeUpdate ();
```

Then generate a new result set from `getGeneratedKeys()` to access the sequence value:

```
long seq;
ResultSet rs = s.getGeneratedKeys ();
if (rs.next ())
{
  seq = rs.getLong (1);
}
else
{
  throw new SQLException ("getGeneratedKeys() produced no value");
}
rs.close ();
s.close ();
```

Server-side and client-side sequence value retrieval compared

As mentioned earlier, the server maintains the value of `LAST_INSERT_ID()` on a session-specific basis. By contrast, the API-specific methods for accessing `AUTO_INCREMENT` values directly are implemented on the client side. Server-side and client-side sequence value retrieval methods have some similarities, but also some differences.

All methods, both server-side and client-side, require that you access an `AUTO_INCREMENT` value within the same MySQL session that generated it. If you generate an `AUTO_INCREMENT` value, then disconnect from the server and reconnect before attempting

to access the value, you'll get zero. Within a given session, the persistence of AUTO_INCREMENT values can be much longer on the server side of the session:

- After you execute a statement that generates an AUTO_INCREMENT value, the value remains available through LAST_INSERT_ID() even if you execute other statements, as long as none of those statements generate an AUTO_INCREMENT value.
- The sequence value available using client-side API methods typically is set for *every* statement, not only those that generate AUTO_INCREMENT values. If you execute an INSERT statement that generates a new value and then execute some other statement before accessing the client-side sequence value, it probably will have been set to zero. The precise behavior varies among APIs, but to be safe, you can do this: when a statement generates a sequence value that you won't use immediately, save the value in a variable that you can refer to later. Otherwise, you may find the sequence value wiped out by the time you try to access it. (For more on this topic, see Recipe 13.10.)

13.5. Renumbering an Existing Sequence

Problem

You have gaps in a sequence column, and you want to resequence it.

Solution

Don't bother. Or at least don't do so without a good reason, of which there are very few.

Discussion

If you insert rows into a table that has an AUTO_INCREMENT column and never delete any of them, values in the column form an unbroken sequence. If you delete rows, the sequence begins to have holes in it. For example, Junior's insect table currently looks something like this, with gaps in the sequence (assuming that you've inserted the cricket and moth rows shown in Recipe 13.4):

```
mysql> SELECT * FROM insect ORDER BY id;
+----+-------------------+------------+------------+
| id | name              | date       | origin     |
+----+-------------------+------------+------------+
|  1 | housefly          | 2014-09-10 | kitchen    |
|  3 | grasshopper       | 2014-09-10 | front yard |
|  4 | stink bug         | 2014-09-10 | front yard |
|  5 | cabbage butterfly | 2014-09-10 | garden     |
|  6 | ant               | 2014-09-10 | back yard  |
|  9 | cricket           | 2014-09-11 | basement   |
```

```
| 10 | moth              | 2014-09-14 | windowsill |
+----+-------------------+------------+------------+
```

MySQL won't attempt to eliminate these gaps by filling in the unused values when you insert new rows. People who dislike this behavior tend to resequence `AUTO_INCREMENT` columns periodically to eliminate the holes. The examples in this section show how to do that. It's also possible to extend the range of an existing sequence (see Recipe 13.6), force deleted values at the top of a sequence to be reused (see Recipe 13.7), number rows in a particular order (see Recipe 13.8), or add a sequence column to a table that doesn't currently have one (see Recipe 13.9).

Before you decide to resequence an `AUTO_INCREMENT` column, consider whether that's really necessary. It usually isn't, and in some cases can cause you real problems. For example, you should *not* resequence a column containing values that are referenced by another table. Renumbering the values destroys their correspondence to values in the other table, making it impossible to properly relate rows in the two tables to each other.

Here are reasons I have seen advanced for resequencing a column:

Aesthetics
: Some people prefer unbroken sequences to sequences with holes in them. If this is why you want to resequence, there's probably not much I can say to convince you otherwise. Nevertheless, it's not a particularly good reason.

Performance
: The impetus for resequencing may stem from the notion that doing so "compacts" a sequence column by removing gaps and enables MySQL to run statements more quickly. This is not true. MySQL doesn't care whether there are holes, and there is no performance gain to be had by renumbering an `AUTO_INCREMENT` column. In fact, resequencing affects performance negatively in the sense that the table remains locked while MySQL performs the operation—which may take a nontrivial amount of time for a large table. Other clients can read from the table while this is happening, but clients trying to insert new rows block until the operation is complete.

Running out of numbers
: The sequence column's data type and signedness determine its upper limit (see Recipe 13.2). If an `AUTO_INCREMENT` sequence is approaching the upper limit of its data type, renumbering packs the sequence and frees up more values at the top. This may be a legitimate reason to resequence a column, but it is still unnecessary in many cases. You may be able to change the column data type to increase its upper limit without changing the values stored in the column; see Recipe 13.6.

If you're still determined to resequence a column, it's easy to do: drop the column from the table; then put it back. MySQL renumbers the values in the column in unbroken

sequence. The following example shows how to renumber the `id` values in the `insect` table using this technique:

```
mysql> ALTER TABLE insect DROP id;
mysql> ALTER TABLE insect
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
    -> ADD PRIMARY KEY (id);
```

The first `ALTER TABLE` statement gets rid of the `id` column (and as a result also drops the `PRIMARY KEY`, because the column to which it refers is no longer present). The second statement restores the column to the table and establishes it as the `PRIMARY KEY`. (The `FIRST` keyword places the column first in the table, which is where it was originally. Normally, `ADD` puts columns at the end of the table.)

When you add an `AUTO_INCREMENT` column to a table, MySQL automatically numbers all the rows consecutively, so the resulting contents of the `insect` table look like this:

```
mysql> SELECT * FROM insect ORDER BY id;
+----+--------------------+------------+------------+
| id | name               | date       | origin     |
+----+--------------------+------------+------------+
|  1 | housefly           | 2014-09-10 | kitchen    |
|  2 | grasshopper        | 2014-09-10 | front yard |
|  3 | stink bug          | 2014-09-10 | front yard |
|  4 | cabbage butterfly | 2014-09-10 | garden     |
|  5 | ant                | 2014-09-10 | back yard  |
|  6 | cricket            | 2014-09-11 | basement   |
|  7 | moth               | 2014-09-14 | windowsill |
+----+--------------------+------------+------------+
```

One problem with resequencing a column using separate `ALTER TABLE` statements is that the table is without that column for the interval between the two operations. This might cause difficulties for other clients that try to access the table during that time. To prevent this from happening, perform both operations with a single `ALTER TABLE` statement:

```
mysql> ALTER TABLE insect
    -> DROP id,
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST;
```

MySQL permits multiple actions to be done with `ALTER TABLE` (something not true for all database systems). However, notice that this multiple-action statement is not simply a concatenation of the two single-action `ALTER TABLE` statements. The difference is that it is unnecessary to reestablish the `PRIMARY KEY`: MySQL doesn't drop it unless the indexed column is missing after all the actions specified in the `ALTER TABLE` statement have been performed.

13.6. Extending the Range of a Sequence Column

Problem

You want to avoid resequencing a column, but you're running out of room for new sequence numbers.

Solution

Check whether you can make the column `UNSIGNED` or change it to use a larger integer type.

Discussion

Resequencing an `AUTO_INCREMENT` column changes the contents of potentially every row in the table. It's often possible to avoid this by extending the range of the column, which changes the table's structure rather than its contents:

- If the data type is signed, make it `UNSIGNED` to double the range of available values. Suppose that an `id` column currently is defined like this:

  ```
  id MEDIUMINT NOT NULL AUTO_INCREMENT
  ```

  The upper range of a signed `MEDIUMINT` column is 8,388,607. To increase this to 16,777,215, make the column `UNSIGNED` with `ALTER TABLE`:

  ```
  ALTER TABLE tbl_name MODIFY id MEDIUMINT UNSIGNED NOT NULL AUTO_INCREMENT;
  ```

- If your column is already `UNSIGNED` and it is not already the largest integer type (`BIGINT`), converting it to a larger type increases its range. Use `ALTER TABLE` for this, too. Convert the `id` column in the previous example from `MEDIUMINT` to `BIGINT` like so:

  ```
  ALTER TABLE tbl_name MODIFY id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT;
  ```

Recipe 13.2 shows the ranges for each integer data type, which can help you choose an appropriate type.

13.7. Reusing Values at the Top of a Sequence

Problem

You've deleted rows at the top end of your sequence. Can you avoid resequencing the column but still reuse the values that have been deleted?

Solution

Yes. Use `ALTER TABLE` to reset the sequence counter. New sequence numbers will begin with the value one larger than the current maximum in the table.

Discussion

If you have removed rows only from the top of the sequence, those that remain are still in order with no gaps. (For example, if you have rows numbered 1 to 100 and you remove the rows with numbers 91 to 100, the remaining rows are still in unbroken sequence from 1 to 90.) In this special case, it's unnecessary to renumber the column. Instead, tell MySQL to resume the sequence beginning with the value one larger than the highest existing sequence number by executing this statement, which causes MySQL to reset the sequence counter down as far as it can for new rows:

```
ALTER TABLE tbl_name AUTO_INCREMENT = 1;
```

You can use `ALTER TABLE` to reset the sequence counter if a sequence column contains gaps in the middle, but doing so still reuses only values deleted from the top of the sequence. It does not eliminate the gaps. Suppose that a table contains sequence values from 1 to 10, from which you delete the rows for values 3, 4, 5, 9, and 10. The maximum remaining value is 8, so if you use `ALTER TABLE` to reset the sequence counter, the next row is given a value of 9, not 3. To resequence a table to eliminate the gaps, see Recipe 13.5.

13.8. Ensuring That Rows Are Renumbered in a Particular Order

Problem

You resequenced a column, but MySQL didn't number the rows the way you want.

Solution

Select the rows into another table, using an `ORDER BY` clause to place them in the order you want, and let MySQL number them according to the sort order as it performs the operation.

Discussion

When you resequence an `AUTO_INCREMENT` column, MySQL is free to pick the rows from the table in any order, so it doesn't necessarily renumber them in the order that you expect. This doesn't matter at all if your only requirement is that each row have a unique identifier. But you might have an application for which it's important that the rows be

assigned sequence numbers in a particular order. For example, you may want the sequence to correspond to the order in which rows were created, as indicated by a `TIMESTAMP` column. To assign numbers in a particular order, use this procedure:

1. Create an empty clone of the table (see Recipe 4.1).
2. Copy rows from the original into the clone using `INSERT INTO ... SELECT`. Copy all columns except the `AUTO_INCREMENT` column, using an `ORDER BY` clause to specify the order in which rows are copied (and thus the order in which MySQL assigns numbers to the `AUTO_INCREMENT` column).
3. Drop the original table and rename the clone to have the original table's name.
4. If the table is a large MyISAM table and has multiple indexes, it is more efficient to create the new table initially with no indexes except the one on the `AUTO_INCREMENT` column. Then copy the original table into the new table and use `ALTER TABLE` to add the remaining indexes afterward.

An alternative procedure:

1. Create a new table that contains all the columns of the original table except the `AUTO_INCREMENT` column.
2. Use `INSERT INTO ... SELECT` to copy the non-`AUTO_INCREMENT` columns from the original table into the new table.
3. Use `TRUNCATE TABLE` on the original table to empty it; this also resets the sequence counter to 1.
4. Copy rows from the new table back to the original table, using an `ORDER BY` clause to sort rows into the order in which you want sequence numbers assigned. MySQL assigns sequence values to the `AUTO_INCREMENT` column.

13.9. Sequencing an Unsequenced Table

Problem

You forgot to include a sequence column when you created a table. Is it too late to sequence the table rows?

Solution

No. Add an `AUTO_INCREMENT` column using `ALTER TABLE`; MySQL creates the column and numbers its rows.

Discussion

To add a sequence to a table that doesn't currently contain one, use `ALTER TABLE` to create an `AUTO_INCREMENT` column. Suppose that a table contains `name` and `age` columns, but no sequence column:

```
mysql> SELECT * FROM t;
+----------+------+
| name     | age  |
+----------+------+
| boris    |   47 |
| clarence |   62 |
| abner    |   53 |
+----------+------+
```

Add a sequence column named `id` to the table as follows:

```
mysql> ALTER TABLE t
    -> ADD id INT NOT NULL AUTO_INCREMENT,
    -> ADD PRIMARY KEY (id);
mysql> SELECT * FROM t ORDER BY id;
+----------+------+----+
| name     | age  | id |
+----------+------+----+
| boris    |   47 |  1 |
| clarence |   62 |  2 |
| abner    |   53 |  3 |
+----------+------+----+
```

MySQL numbers the rows for you; it's unnecessary to assign the values yourself. Very handy.

By default, `ALTER TABLE` adds new columns to the end of the table. To place a column at a specific position, use `FIRST` or `AFTER` at the end of the `ADD` clause. The following `ALTER TABLE` statements are similar to the one just shown, but place the `id` column first in the table or after the `name` column, respectively:

```
ALTER TABLE t
  ADD id INT NOT NULL AUTO_INCREMENT FIRST,
  ADD PRIMARY KEY (id);

ALTER TABLE t
  ADD id INT NOT NULL AUTO_INCREMENT AFTER name,
  ADD PRIMARY KEY (id);
```

13.10. Managing Multiple Auto-Increment Values Simultaneously

Problem

You're executing multiple statements that generate AUTO_INCREMENT values, and it's necessary to keep track of them independently. For example, you're inserting rows into multiple tables, each of which has its own AUTO_INCREMENT column.

Solution

Save the sequence values in variables for later use. Alternatively, if you execute sequence-generating statements from within a program, you might be able to issue the statements using separate connection or statement objects to keep them from getting mixed up.

Discussion

As described in Recipe 13.4, the LAST_INSERT_ID() server-side sequence value function is set each time a statement generates an AUTO_INCREMENT value, whereas client-side sequence indicators may be reset for every statement. What if you issue a statement that generates an AUTO_INCREMENT value, but you don't want to refer to that value until after issuing a second statement that also generates an AUTO_INCREMENT value? In this case, the original value is no longer accessible, either through LAST_INSERT_ID() or as a client-side value. To retain access to it, save the value first before issuing the second statement. There are several ways to do this:

- At the SQL level, save the value in a user-defined variable after issuing a statement that generates an AUTO_INCREMENT value:

  ```
  INSERT INTO tbl_name (id,...) VALUES(NULL,...);
  SET @saved_id = LAST_INSERT_ID();
  ```

  Then you can issue other statements without regard to their effect on LAST_INSERT_ID(). To use the original AUTO_INCREMENT value in a subsequent statement, refer to the @saved_id variable.
- At the API level, save the AUTO_INCREMENT value in an API language variable. This can be done by saving the value returned from either LAST_INSERT_ID() or any API-specific extension that is available.
- Some APIs enable you to maintain separate client-side AUTO_INCREMENT values. For example, Perl DBI statement handles have a mysql_insertid attribute, and the attribute value for one handle is unaffected by activity on another. In Java, use separate Statement or PreparedStatement objects.

See Recipe 13.11 for application of these techniques to situations in which you must insert rows into multiple tables that each contain an AUTO_INCREMENT column.

13.11. Using Auto-Increment Values to Associate Tables

Problem

You use sequence values from one table as keys in a second table so that you can associate rows in the two tables with each other. But the associations aren't being set up properly.

Solution

You're probably not inserting rows in the proper order, or you're losing track of the sequence values. Change the insertion order, or save the sequence values so that you can refer to them when you need them.

Discussion

Be careful with an AUTO_INCREMENT value used as an ID value in a master table if you also store the value in detail table rows for the purpose of linking the detail rows to the proper master table row. Suppose that an `invoice` table lists invoice information for customer orders, and an `inv_item` table lists the individual items associated with each invoice. Here, `invoice` is the master table and `inv_item` is the detail table. To uniquely identify each order, include an AUTO_INCREMENT column `inv_id` in the `invoice` table. You'd also store the appropriate invoice number in each `inv_item` table row so that you can tell which invoice it goes with. The tables might look something like this:

```
CREATE TABLE invoice
(
  inv_id  INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (inv_id),
  date    DATE NOT NULL
  # ... other columns could go here
  # ... (customer ID, shipping address, etc.)
);
CREATE TABLE inv_item
(
  inv_id    INT UNSIGNED NOT NULL,  # invoice ID (from invoice table)
  INDEX (inv_id),
  qty       INT,                    # quantity
  description VARCHAR(40)           # description
);
```

For this kind of table relationship, it's typical to insert a row into the master table first (to generate the AUTO_INCREMENT value that identifies the row), and then insert the detail rows using `LAST_INSERT_ID()` to obtain the master row ID. If a customer buys a hammer, three boxes of nails, and (in anticipation of finger-bashing with the hammer) a

dozen bandages, the rows pertaining to the order can be inserted into the two tables like so:

```
INSERT INTO invoice (inv_id,date)
  VALUES(NULL,CURDATE());
INSERT INTO inv_item (inv_id,qty,description)
  VALUES(LAST_INSERT_ID(),1,'hammer');
INSERT INTO inv_item (inv_id,qty,description)
  VALUES(LAST_INSERT_ID(),3,'nails, box');
INSERT INTO inv_item (inv_id,qty,description)
  VALUES(LAST_INSERT_ID(),12,'bandage');
```

The first `INSERT` adds a row to the `invoice` master table and generates a new `AUTO_INCREMENT` value for its `inv_id` column. The following `INSERT` statements each add a row to the `inv_item` detail table, using `LAST_INSERT_ID()` to get the invoice number. This associates the detail rows with the proper master row.

What if you have multiple invoices to process? There's a right way and a wrong way to enter the information. The right way is to insert all the information for the first invoice, then proceed to the next. The wrong way is to add all the master rows into the `invoice` table, then add all the detail rows to the `inv_item` table. If you do that, *all* the new detail rows in the `inv_item` table have the `AUTO_INCREMENT` value from the most recently entered `invoice` row. Thus, all items appear to be part of that invoice, and rows in the two tables don't have the proper associations.

If the detail table contains its own `AUTO_INCREMENT` column, you must be even more careful about how you add rows to the tables. Suppose that you want each row in the `inv_item` table to have a unique identifier. To do that, create the `inv_item` table as follows with an `AUTO_INCREMENT` column named `item_id`:

```
CREATE TABLE inv_item
(
  inv_id  INT UNSIGNED NOT NULL,  # invoice ID (from invoice table)
  item_id INT UNSIGNED NOT NULL AUTO_INCREMENT, # item ID
  PRIMARY KEY (item_id),
  qty     INT,                                  # quantity
  description VARCHAR(40)                       # description
);
```

The `inv_id` column enables each `inv_item` row to be associated with the proper `invoice` table row, just as with the original table structure. In addition, `item_id` uniquely identifies each item row. However, now that both tables contain an `AUTO_INCREMENT` column, you cannot enter information for an invoice the same way as before. If you execute the `INSERT` statements shown previously, they now produce a different result due to the change in the `inv_item` table structure. The `INSERT` into the `invoice` table works properly. So does the first `INSERT` into the `inv_item` table; `LAST_INSERT_ID()` returns the `inv_id` value from the master row in the `invoice` table. However, this `INSERT` also generates its own `AUTO_INCREMENT` value (for the `item_id` column), which changes

the value of `LAST_INSERT_ID()` and causes the master row `inv_id` value to be "lost." As a result, each of the remaining inserts into the `inv_item` table stores the preceding row's `item_id` value into the `inv_id` column. This causes the second and following rows to have incorrect `inv_id` values.

To avoid this difficulty, save the sequence value generated by the insert into the master table and use the saved value for the inserts into the detail table. To save the value, use a user-defined SQL variable or a variable maintained by your program. Recipe 13.10 describes those techniques, which apply here as follows:

- Use a user-defined variable: Save the master row `AUTO_INCREMENT` value in a user-defined variable for use when inserting the detail rows:

```
INSERT INTO invoice (inv_id,date)
  VALUES(NULL,CURDATE());
SET @inv_id = LAST_INSERT_ID();
INSERT INTO inv_item (inv_id,qty,description)
  VALUES(@inv_id,1,'hammer');
INSERT INTO inv_item (inv_id,qty,description)
  VALUES(@inv_id,3,'nails, box');
INSERT INTO inv_item (inv_id,qty,description)
  VALUES(@inv_id,12,'bandage');
```

- Use a variable maintained by your program: This method is similar to the previous one, but applies only from within an API. Insert the master row, and then save the `AUTO_INCREMENT` value into an API variable for use when inserting detail rows. For example, in Ruby, access the `AUTO_INCREMENT` value using the `insert_id` database-handle attribute:

```
dbh.do("INSERT INTO invoice (inv_id,date) VALUES(NULL,CURDATE())")
inv_id = dbh.func(:insert_id)
sth = dbh.prepare("INSERT INTO inv_item (inv_id,qty,description)
                   VALUES(?,?,?)")
sth.execute(inv_id, 1, "hammer")
sth.execute(inv_id, 3, "nails, box")
sth.execute(inv_id, 12, "bandage")
```

13.12. Using Sequence Generators as Counters

Problem

You're interested only in counting events, so there's no point in creating a table row for each sequence value.

Solution

Use a sequence-generation mechanism that uses a single row per counter.

Discussion

AUTO_INCREMENT columns are useful for generating sequences across a set of individual rows. But some applications require only a count of the number of times an event occurs, and there's no benefit from creating a separate row for each event. Instances include web page or banner ad hit counters, a count of items sold, or the number of votes in a poll. Such applications need only a single row to hold the count as it changes over time. MySQL provides a mechanism for this that enables counts to be treated like AUTO_INCREMENT values so that you can not only increment the count, but retrieve the updated value easily.

To count a single type of event, use a trivial table with a single row and column. For example, to record copies sold of a book, create a table like this:

```
CREATE TABLE booksales (copies INT UNSIGNED);
```

However, if you're counting sales for multiple book titles, that method doesn't work well. You certainly don't want to create a separate single-row counting table per book. Instead, count them all within a single table by including a column that uniquely identifies each book. The following table does this using a `title` column for the book title in addition to a `copies` column that records the number of copies sold:

```
CREATE TABLE booksales
(
  title   VARCHAR(60) NOT NULL,   # book title
  copies  INT UNSIGNED NOT NULL,  # number of copies sold
  PRIMARY KEY (title)
);
```

To record sales for a given book, different approaches are possible:

- Initialize a row for the book with a `copies` value of 0:

  ```
  INSERT INTO booksales (title,copies) VALUES('The Greater Trumps',0);
  ```

  Then increment the `copies` value for each sale:

  ```
  UPDATE booksales SET copies = copies+1 WHERE title = 'The Greater Trumps';
  ```

  This method requires that you remember to initialize a row for each book or the UPDATE will fail.

- Use INSERT with ON DUPLICATE KEY UPDATE, which initializes the row with a count of 1 for the first sale and increments the count for subsequent sales:

  ```
  INSERT INTO booksales (title,copies)
  VALUES('The Greater Trumps',1)
  ON DUPLICATE KEY UPDATE copies = copies+1;
  ```

  This is simpler because the same statement works to initialize and update the sales count.

To retrieve the sales count (for example, to display a message to customers such as "you just purchased copy *n* of this book"), issue a `SELECT` query for the same book title:

```
SELECT copies FROM booksales WHERE title = 'The Greater Trumps';
```

Unfortunately, this is not quite correct. Suppose that between the times when you update and retrieve the count, some other person buys a copy of the book (and thus increments the `copies` value). Then the `SELECT` statement won't actually produce the value *you* incremented the sales count to, but rather its most recent value. In other words, other clients can affect the value before you have time to retrieve it. This is similar to the problem discussed in Recipe 13.4 that can occur if you try to retrieve the most recent `AUTO_INCREMENT` value from a column by invoking `MAX(`*`col_name`*`)` rather than `LAST_INSERT_ID()`.

There are ways around this (such as by grouping the two statements as a transaction or by locking the table), but MySQL provides a simpler solution based on `LAST_INSERT_ID()`. If you call `LAST_INSERT_ID()` with an expression argument, MySQL treats it like an `AUTO_INCREMENT` value. To use this feature with the `booksales` table, modify the count-incrementing statement slightly:

```
INSERT INTO booksales (title,copies)
VALUES('The Greater Trumps',LAST_INSERT_ID(1))
ON DUPLICATE KEY UPDATE copies = LAST_INSERT_ID(copies+1);
```

The statement uses the `LAST_INSERT_ID(`*`expr`*`)` construct both to initialize and to increment the count. MySQL treats the expression argument like an `AUTO_INCREMENT` value, so that you can invoke `LAST_INSERT_ID()` later with no argument to retrieve the value:

```
SELECT LAST_INSERT_ID();
```

By setting and retrieving the `copies` column this way, you always get back the value you set it to, even if some other client updated it in the meantime. If you issue the `INSERT` statement from within an API that provides a mechanism for fetching the most recent `AUTO_INCREMENT` value directly, you need not even issue the `SELECT` query. For example, using Connector/Python, update a count and get the new value using the `lastrowid` attribute:

```
cursor = conn.cursor()
cursor.execute('''
               INSERT INTO booksales (title,copies)
               VALUES('The Greater Trumps',LAST_INSERT_ID(1))
               ON DUPLICATE KEY UPDATE copies = LAST_INSERT_ID(copies+1)
               ''')
count = cursor.lastrowid
cursor.close()
conn.commit()
```

In Java, the operation looks like this:

```
Statement s = conn.createStatement ();
s.executeUpdate (
    "INSERT INTO booksales (title,copies)"
    + "VALUES('The Greater Trumps',LAST_INSERT_ID(1))"
    + "ON DUPLICATE KEY UPDATE copies = LAST_INSERT_ID(copies+1)",
    Statement.RETURN_GENERATED_KEYS);
long count;
ResultSet rs = s.getGeneratedKeys ();
if (rs.next ())
{
  count = rs.getLong (1);
}
else
{
  throw new SQLException ("getGeneratedKeys() produced no value");
}
rs.close ();
s.close ();
```

Use of `LAST_INSERT_ID(`*`expr`*`)` for sequence generation has certain other properties that differ from true `AUTO_INCREMENT` sequences:

- `AUTO_INCREMENT` values increment by one each time, whereas values generated by `LAST_INSERT_ID(`*`expr`*`)` can be any nonnegative value you want. For example, to produce the sequence 10, 20, 30, …, increment the count by 10 each time. You need not even increment the counter by the same value each time. If you sell a dozen copies of a book rather than a single copy, update its sales count as follows:

  ```
  INSERT INTO booksales (title,copies)
  VALUES('The Greater Trumps',LAST_INSERT_ID(12))
  ON DUPLICATE KEY UPDATE copies = LAST_INSERT_ID(copies+12);
  ```

- To reset a counter, simply set it to the desired value. Suppose that you want to report to book buyers the sales for the current month, rather than the total sales (for example, to display messages like "you're the *n*th buyer this month"). To clear the counters to zero at the beginning of each month, use this statement:

  ```
  UPDATE booksales SET copies = 0;
  ```

- One property that's not so desirable is that the value generated by `LAST_INSERT_ID(`*`expr`*`)` is not uniformly available via client-side retrieval methods under all circumstances. You can get it after `UPDATE` or `INSERT` statements, but not for `SET` statements. If you generate a value as follows (in Ruby), the client-side value returned by `insert_id` is 0, not 48:

  ```
  dbh.do("SET @x = LAST_INSERT_ID(48)")
  seq = dbh.func(:insert_id)
  ```

  To get the value in this case, ask the server for it:

  ```
  seq = dbh.select_one("SELECT LAST_INSERT_ID()")[0]
  ```

See Also

Recipe 20.12 uses this single-row sequence-generation mechanism as the basis for implementing web page hit counters.

13.13. Generating Repeating Sequences

Problem

You require a sequence that contains cycles.

Solution

Generate a sequence and use it to produce cyclic elements with division and modulo operations.

Discussion

Some sequence-generation problems require values that go through cycles. Suppose that you manufacture items such as pharmaceutical products or automobile parts, and you must be able to track them by lot number if manufacturing problems are discovered later that require items sold within a particular lot to be recalled. Suppose also that you pack and distribute items 12 units to a box and 6 boxes to a case. In this situation, item identifiers are three-part values: the unit number (with a value from 1 to 12), the box number (with a value from 1 to 6), and a lot number (with a value from 1 to the highest current case number).

This item-tracking problem appears to require that you maintain three counters, so you might generate the next identifier value using an algorithm like this:

```
retrieve most recently used case, box, and unit numbers
unit = unit + 1     # increment unit number
if (unit > 12)      # need to start a new box?
{
  unit = 1          # go to first unit of next box
  box = box + 1
}
if (box > 6)        # need to start a new case?
{
  box = 1           # go to first box of next case
  case = case + 1
}
store new case, box, and unit numbers
```

Alternatively, it's possible simply to assign each item a sequence number identifier and derive the corresponding case, box, and unit numbers from it. The identifier can come from an `AUTO_INCREMENT` column or a single-row sequence generator. The formulas for

determining the case, box, and unit numbers for any item from its sequence number look like this:

```
unit_num = ((seq - 1) % 12) + 1
box_num = (int ((seq - 1) / 12) % 6) + 1
case_num = int ((seq - 1)/(6 * 12)) + 1
```

The following table illustrates the relationship between some sample sequence numbers and the corresponding case, box, and unit numbers:

seq	case	box	unit
1	1	1	1
12	1	1	12
13	1	2	1
72	1	6	12
73	2	1	1
144	2	6	12

CHAPTER 14
Using Joins and Subqueries

14.0. Introduction

Most queries in earlier chapters used a single table, but for any application of even moderate complexity, you'll likely need to use multiple tables. Some questions simply cannot be answered using a single table, and the real power of a relational database comes into play when you combine the information from multiple sources:

- To combine rows from tables to obtain more comprehensive information than can be obtained from individual tables alone
- To hold intermediate results for a multiple-stage operation
- To modify rows in one table based on information from another

This chapter focuses on two types of statements that use multiple tables: joins between tables and subqueries that nest one `SELECT` within another. It covers the following topics:

Comparing tables to find matches or mismatches
: To solve such problems, you should know which types of joins apply. Inner joins show which rows in one table match rows in another. Outer joins show matching rows, but also find rows in one table *not* matched by rows in another.

Deleting unmatched rows
: If two datasets are related, but imperfectly, you can determine which rows are unmatched and remove them as necessary.

Comparing a table to itself
: Some problems require comparing a table to itself. This is similar to performing a join between different tables, except that you must use table aliases to disambiguate table references.

Producing master-detail and many-to-many relationships
: Joins enable production of lists or summaries when each item in one table can match many in the other, or when each item in either table can match many in the other.

Scripts that create tables used in this chapter are located in the *tables* directory of the `recipes` distribution. For scripts that implement techniques discussed here, look in the *joins* directory.

14.1. Finding Matches Between Tables

Problem

You need to perform a task that requires information from more than one table.

Solution

Use a join—that is, a query that lists multiple tables in its `FROM` clause and tells MySQL how to match information from them.

Discussion

The essential idea behind a join is that it matches rows in one table with rows in one or more other tables. Joins enable you to combine information from multiple tables when each one answers only part of the question in which you're interested.

A complete join that produces all possible row combinations is called a Cartesian product. For example, joining each row in a 100-row table to each row in a 200-row table produces a result containing 100 × 200 = 20,000 rows. With larger tables, or joins between more than two tables, the result set for a Cartesian product easily becomes immense, so a join normally includes an `ON` or `USING` comparison clause to produce only the desired matches between tables. (This requires that each table have one or more columns of common information that link them together logically.) You can also include a `WHERE` clause that restricts which of the joined rows to select. Each clause narrows the focus of the query.

This section introduces join syntax and demonstrates how joins answer specific types of questions when you are looking for matches between tables. Other sections show how to identify mismatches between tables (see Recipe 14.2) and how to compare a table to itself (see Recipe 14.4). The examples assume that you have an art collection and use the following two tables to record your acquisitions. `artist` lists those painters whose works you want to collect, and `painting` lists each painting you've actually purchased:

```
CREATE TABLE artist
(
  a_id  INT UNSIGNED NOT NULL AUTO_INCREMENT, # artist ID
  name  VARCHAR(30) NOT NULL,                 # artist name
```

```
  PRIMARY KEY (a_id),
  UNIQUE (name)
);

CREATE TABLE painting
(
  a_id  INT UNSIGNED NOT NULL,                # artist ID
  p_id  INT UNSIGNED NOT NULL AUTO_INCREMENT, # painting ID
  title VARCHAR(100) NOT NULL,                # title of painting
  state VARCHAR(2) NOT NULL,                  # state where purchased
  price INT UNSIGNED,                         # purchase price (dollars)
  INDEX (a_id),
  PRIMARY KEY (p_id)
);
```

You've just begun the collection, so the tables contain only a few rows:

```
mysql> SELECT * FROM artist ORDER BY a_id;
+------+----------+
| a_id | name     |
+------+----------+
|    1 | Da Vinci |
|    2 | Monet    |
|    3 | Van Gogh |
|    4 | Renoir   |
+------+----------+
mysql> SELECT * FROM painting ORDER BY a_id, p_id;
+------+------+------------------+-------+-------+
| a_id | p_id | title            | state | price |
+------+------+------------------+-------+-------+
|    1 |    1 | The Last Supper  | IN    |    34 |
|    1 |    2 | Mona Lisa        | MI    |    87 |
|    3 |    3 | Starry Night     | KY    |    48 |
|    3 |    4 | The Potato Eaters | KY    |    67 |
|    4 |    5 | Les Deux Soeurs  | NE    |    64 |
+------+------+------------------+-------+-------+
```

The low values in the `price` column of the `painting` table betray the fact that your collection actually contains only cheap imitations, not the originals. Well, that's all right: who can afford the originals?

Each table contains partial information about your collection. For example, the `artist` table doesn't tell you which paintings each artist produced, and the `painting` table lists artist IDs but not their names. To use the information in both tables, write a query that performs a join. A join names two or more tables after the `FROM` keyword. In the output column list, use `*` to select all columns from all tables, *`tbl_name`*`.*` to select all columns from a given table, or name specific columns from the joined tables or expressions based on those columns.

The simplest join involves two tables and selects all columns from each. The following join between the `artist` and `painting` tables shows this (the `ORDER BY` clause makes the result easier to read):

```
mysql> SELECT * FROM artist INNER JOIN painting ORDER BY artist.a_id;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    3 |    3 | Starry Night      | KY    |    48 |
|    1 | Da Vinci |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
|    1 | Da Vinci |    1 |    2 | Mona Lisa         | MI    |    87 |
|    1 | Da Vinci |    3 |    4 | The Potato Eaters | KY    |    67 |
|    2 | Monet    |    1 |    2 | Mona Lisa         | MI    |    87 |
|    2 | Monet    |    3 |    4 | The Potato Eaters | KY    |    67 |
|    2 | Monet    |    1 |    1 | The Last Supper   | IN    |    34 |
|    2 | Monet    |    3 |    3 | Starry Night      | KY    |    48 |
|    2 | Monet    |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
|    3 | Van Gogh |    1 |    2 | Mona Lisa         | MI    |    87 |
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    3 | Van Gogh |    1 |    1 | The Last Supper   | IN    |    34 |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
|    4 | Renoir   |    1 |    1 | The Last Supper   | IN    |    34 |
|    4 | Renoir   |    3 |    3 | Starry Night      | KY    |    48 |
|    4 | Renoir   |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
|    4 | Renoir   |    1 |    2 | Mona Lisa         | MI    |    87 |
|    4 | Renoir   |    3 |    4 | The Potato Eaters | KY    |    67 |
+------+----------+------+------+-------------------+-------+-------+
```

An `INNER JOIN` produces results that combine values in one table with values in another table. The preceding query specifies no restrictions on row matching, so the join generates all row combinations (that is, the Cartesian product). This result illustrates why such a join generally is not useful: it produces a lot of unmeaningful output. Clearly, you don't maintain these tables to match every artist with every painting.

To answer questions meaningfully, produce only the relevant matches by including appropriate join conditions. For example, to produce a list of paintings together with the artist names, associate rows from the two tables using a simple `WHERE` clause that matches values based on the artist ID column that is common to both tables and serves to link them:

```
mysql> SELECT * FROM artist INNER JOIN painting
    -> WHERE artist.a_id = painting.a_id
    -> ORDER BY artist.a_id;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    1 |    2 | Mona Lisa         | MI    |    87 |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
```

```
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    4 | Renoir   |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+------+-------------------+-------+-------+
```

The column names in the `WHERE` clause include table qualifiers to make it clear which `a_id` values to compare. The result indicates who painted each painting, and, conversely, which paintings by each artist are in your collection.

Joins and Indexes

A join can easily cause MySQL to process large numbers of row combinations, so it's a good idea to index the comparison columns. Otherwise, performance drops off quickly as table sizes increase. For the `artist` and `painting` tables, joins are made by comparing the `a_id` columns. If you look back at the `CREATE TABLE` statements for those tables, you see that `a_id` is indexed in each table.

Another way to write the same join indicates the matching conditions with an `ON` clause:

```
SELECT * FROM artist INNER JOIN painting
ON artist.a_id = painting.a_id
ORDER BY artist.a_id;
```

In the special case of equality comparisons between columns with the same name in both tables, you can use an `INNER JOIN` with a `USING` clause instead. This requires no table qualifiers and names each joined column only once:

```
SELECT * FROM artist INNER JOIN painting
USING (a_id)
ORDER BY a_id;
```

For `SELECT *` queries, the `USING` form produces a result that differs from the `ON` form: it returns only one instance of each join column, so `a_id` appears once, not twice.

Any of `ON`, `USING`, or `WHERE` can include comparisons, so how do you know which join conditions to put in each clause? As a rule of thumb, it's conventional to use `ON` or `USING` to specify how to join the tables, and the `WHERE` clause to restrict which of the joined rows to select. For example, to join tables based on the `a_id` column, but select only rows for paintings obtained in Kentucky, use an `ON` (or `USING`) clause to match the rows in the two tables, and a `WHERE` clause to test the `state` column:

```
mysql> SELECT * FROM artist INNER JOIN painting
    -> ON artist.a_id = painting.a_id
    -> WHERE painting.state = 'KY';
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
```

```
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
+------+----------+------+------+-------------------+-------+-------+
```

The preceding queries use `SELECT *` to display all columns. To be more selective, name only those columns in which you're interested:

```
mysql> SELECT artist.name, painting.title, painting.state, painting.price
    -> FROM artist INNER JOIN painting
    -> ON artist.a_id = painting.a_id
    -> WHERE painting.state = 'KY';
+----------+-------------------+-------+-------+
| name     | title             | state | price |
+----------+-------------------+-------+-------+
| Van Gogh | Starry Night      | KY    |    48 |
| Van Gogh | The Potato Eaters | KY    |    67 |
+----------+-------------------+-------+-------+
```

Joins can use more than two tables. Suppose that you prefer to see complete state names rather than abbreviations in the preceding query result. The `states` table used in earlier chapters maps state abbreviations to names; add it to the previous query to display name rather than abbreviation:

```
mysql> SELECT artist.name, painting.title, states.name, painting.price
    -> FROM artist INNER JOIN painting INNER JOIN states
    -> ON artist.a_id = painting.a_id AND painting.state = states.abbrev
    -> WHERE painting.state = 'KY';
+----------+-------------------+----------+-------+
| name     | title             | name     | price |
+----------+-------------------+----------+-------+
| Van Gogh | Starry Night      | Kentucky |    48 |
| Van Gogh | The Potato Eaters | Kentucky |    67 |
+----------+-------------------+----------+-------+
```

Another common use of three-way joins is enumerating many-to-many relationships (see Recipe 14.6).

By including appropriate conditions in your joins, you can answer very specific questions:

- Which paintings did Van Gogh paint? Use the `a_id` value to find matching rows, add a `WHERE` clause to restrict output to rows that contain the artist name, and select the title from those rows:

```
mysql> SELECT painting.title
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> WHERE artist.name = 'Van Gogh';
+-------------------+
| title             |
+-------------------+
| Starry Night      |
| The Potato Eaters |
+-------------------+
```

- Who painted the *Mona Lisa*? Again, use the `a_id` column to join the rows, but this time use the `WHERE` clause to restrict output to rows that contain the title, and select the artist name from those rows:

```
mysql> SELECT artist.name
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> WHERE painting.title = 'Mona Lisa';
+----------+
| name     |
+----------+
| Da Vinci |
+----------+
```

- For which artists did you purchase paintings in Kentucky or Indiana? This is similar to the previous statement, but tests a different column in the `painting` table (`state`) to restrict output to rows for `KY` or `IN`:

```
mysql> SELECT DISTINCT artist.name
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> WHERE painting.state IN ('KY','IN');
+----------+
| name     |
+----------+
| Da Vinci |
| Van Gogh |
+----------+
```

  The statement also uses `DISTINCT` to display each artist name just once. Try it without `DISTINCT`; Van Gogh appears twice because you obtained two Van Goghs in Kentucky.

- Joins used with aggregate functions produce summaries. This statement shows how many paintings you have per artist:

```
mysql> SELECT artist.name, COUNT(*) AS 'number of paintings'
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
+----------+---------------------+
| name     | number of paintings |
+----------+---------------------+
| Da Vinci |                   2 |
| Renoir   |                   1 |
| Van Gogh |                   2 |
+----------+---------------------+
```

  A more elaborate statement uses aggregates to also show how much you paid for each artist's paintings, in total and on average:

```
mysql> SELECT artist.name,
    -> COUNT(*) AS 'number of paintings',
    -> SUM(painting.price) AS 'total price',
    -> AVG(painting.price) AS 'average price'
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
```

```
+----------+--------------------+-------------+---------------+
| name     | number of paintings | total price | average price |
+----------+--------------------+-------------+---------------+
| Da Vinci |                  2 |         121 |       60.5000 |
| Renoir   |                  1 |          64 |       64.0000 |
| Van Gogh |                  2 |         115 |       57.5000 |
+----------+--------------------+-------------+---------------+
```

The preceding summary statements produce output only for those artists in the `artist` table for whom you actually have acquired paintings. (For example, Monet is listed in the `artist` table but is not present in the summary because you have none of his paintings yet.) To summarize all artists, including those for whom you have no paintings, you must use a different kind of join—specifically, an outer join:

- Joins written with `INNER JOIN` are inner joins. They produce a result only for values in one table that match values in another table.
- An outer join can produce those matches as well, but also can show you which values in one table are missing from the other. Recipe 14.2 introduces outer joins.

The *`tbl_name.col_name`* notation that qualifies a column name with a table name is always permitted in a join but can be shortened to just *`col_name`* if the name appears in only one of the joined tables. In that case, MySQL can determine without ambiguity which table the column comes from, and no table name qualifier is necessary. We can't do that for the following join. Both tables have an `a_id` column, so the `ON` clause column references are ambiguous:

```
mysql> SELECT * FROM artist INNER JOIN painting ON a_id = a_id;
ERROR 1052 (23000): Column 'a_id' in on clause is ambiguous
```

By contrast, the following query is unambiguous. Each instance of `a_id` is qualified with the appropriate table name, only `artist` has a `name` column, and only `painting` has `title` and `state` columns:

```
mysql> SELECT name, title, state FROM artist INNER JOIN painting
    -> ON artist.a_id = painting.a_id
    -> ORDER BY name;
+----------+-------------------+-------+
| name     | title             | state |
+----------+-------------------+-------+
| Da Vinci | The Last Supper   | IN    |
| Da Vinci | Mona Lisa         | MI    |
| Renoir   | Les Deux Soeurs   | NE    |
| Van Gogh | Starry Night      | KY    |
| Van Gogh | The Potato Eaters | KY    |
+----------+-------------------+-------+
```

To make the meaning of a statement clearer to human readers, it's often useful to qualify column names even when that's not strictly necessary as far as MySQL is concerned. I tend to use qualified names in join examples for that reason.

To avoid writing complete table names when qualifying column references, give each table a short alias and refer to its columns using the alias. The following two statements are equivalent:

```
SELECT artist.name, painting.title, states.name, painting.price
FROM artist INNER JOIN painting INNER JOIN states
ON artist.a_id = painting.a_id AND painting.state = states.abbrev;

SELECT a.name, p.title, s.name, p.price
FROM artist AS a INNER JOIN painting AS p INNER JOIN states AS s
ON a.a_id = p.a_id AND p.state = s.abbrev;
```

In AS *alias_name* clauses, the AS is optional.

For complicated statements that select many columns, aliases can save a lot of typing. In addition, for some types of statements, aliases are not only convenient but necessary, as will become evident when we get to the topic of self-joins (see Recipe 14.4).

Joining Tables from Different Databases

To perform a join between tables located in different databases, qualify table and column names sufficiently that MySQL knows what you're referring to. Thus far, we have used the `artist` and `painting` tables with the implicit understanding that both are in the `cookbook` database, so we can simply refer to the tables without specifying any database name when `cookbook` is the default database. For example, the following statement uses the two tables to associate artists with their paintings:

```
SELECT artist.name, painting.title
FROM artist INNER JOIN painting
ON artist.a_id = painting.a_id;
```

But suppose instead that `artist` is in the `db1` database and `painting` is in the `db2` database. To indicate this, qualify each table name with a prefix that specifies which database it's in. The fully qualified form of the join looks like this:

```
SELECT db1.artist.name, db2.painting.title
FROM db1.artist INNER JOIN db2.painting
ON db1.artist.a_id = db2.painting.a_id;
```

Table aliases can simplify that considerably:

```
SELECT a.name, p.title
FROM db1.artist AS a INNER JOIN db2.painting AS p
ON a.a_id = p.a_id;
```

If there is no default database, or it is neither `db1` nor `db2`, it's necessary to fully qualify both table names. If the default database is either `db1` or `db2`, you can dispense with the

corresponding qualifiers. If the default database is `db1`, you can omit the `db1` qualifiers. Conversely, if the default database is `db2`, no `db2` qualifiers are necessary.

14.2. Finding Mismatches Between Tables

Problem

You want to find rows in one table that have no match in another. Or you want to produce a list on the basis of a join between tables, and you want the list to include an entry for every row in the first table, including those for which no match occurs in the second table.

Solution

Use an outer join (a `LEFT JOIN` or a `RIGHT JOIN`) or a `NOT IN` subquery.

Discussion

Recipe 14.1 focuses on inner joins, which find matches between two tables. However, the answers to some questions require determining which rows do *not* have a match (or, stated another way, which rows have values missing from the other table). For example, you might want to know artists in the `artist` table for whom you have no paintings yet. Similar questions occur in other contexts:

- You have a list of potential customers, and another list of people who have placed orders. To focus sales efforts on people who are not yet actual customers, produce the set of people who are in the first list but not the second.
- You have one list of baseball players, and another list of players who have hit home runs. To determine which players in the first list have *not* hit a home run, produce the set of players who are in the first list but not the second.

These types of questions require use of an outer join. Like inner joins, an outer join finds matches between tables. But unlike an inner join, an outer join also determines which rows in one table have no match in another. Two types of outer join are `LEFT JOIN` and `RIGHT JOIN`.

To see how outer joins are useful, consider the problem of determining which artists in the `artist` table are missing from the `painting` table. At present, the tables are small, so it's easy to examine them visually and see that you have no paintings by Monet (there are no `painting` rows with an `a_id` value of 2):

```
mysql> SELECT * FROM artist ORDER BY a_id;
+------+----------+
| a_id | name     |
+------+----------+
|    1 | Da Vinci |
|    2 | Monet    |
|    3 | Van Gogh |
|    4 | Renoir   |
+------+----------+
mysql> SELECT * FROM painting ORDER BY a_id, p_id;
+------+------+------------------+-------+-------+
| a_id | p_id | title            | state | price |
+------+------+------------------+-------+-------+
|    1 |    1 | The Last Supper  | IN    |    34 |
|    1 |    2 | Mona Lisa        | MI    |    87 |
|    3 |    3 | Starry Night     | KY    |    48 |
|    3 |    4 | The Potato Eaters| KY    |    67 |
|    4 |    5 | Les Deux Soeurs  | NE    |    64 |
+------+------+------------------+-------+-------+
```

As you acquire more paintings and the tables get larger, it won't be so easy to eyeball them and answer questions by inspection. Can you answer it using SQL? Sure, although first attempts at a solution often look something like the following statement, which uses a not-equal condition to look for mismatches between the two tables:

```
mysql> SELECT * FROM artist INNER JOIN painting
    -> ON artist.a_id <> painting.a_id
    -> ORDER BY artist.a_id;
+------+----------+------+------+------------------+-------+-------+
| a_id | name     | a_id | p_id | title            | state | price |
+------+----------+------+------+------------------+-------+-------+
|    1 | Da Vinci |    4 |    5 | Les Deux Soeurs  | NE    |    64 |
|    1 | Da Vinci |    3 |    4 | The Potato Eaters| KY    |    67 |
|    1 | Da Vinci |    3 |    3 | Starry Night     | KY    |    48 |
|    2 | Monet    |    1 |    1 | The Last Supper  | IN    |    34 |
|    2 | Monet    |    4 |    5 | Les Deux Soeurs  | NE    |    64 |
|    2 | Monet    |    3 |    4 | The Potato Eaters| KY    |    67 |
|    2 | Monet    |    3 |    3 | Starry Night     | KY    |    48 |
|    2 | Monet    |    1 |    2 | Mona Lisa        | MI    |    87 |
|    3 | Van Gogh |    1 |    2 | Mona Lisa        | MI    |    87 |
|    3 | Van Gogh |    1 |    1 | The Last Supper  | IN    |    34 |
|    3 | Van Gogh |    4 |    5 | Les Deux Soeurs  | NE    |    64 |
|    4 | Renoir   |    3 |    3 | Starry Night     | KY    |    48 |
|    4 | Renoir   |    1 |    2 | Mona Lisa        | MI    |    87 |
|    4 | Renoir   |    1 |    1 | The Last Supper  | IN    |    34 |
|    4 | Renoir   |    3 |    4 | The Potato Eaters| KY    |    67 |
+------+----------+------+------+------------------+-------+-------+
```

The query may look plausible but its result obviously is not. For example, it falsely indicates that each painting was painted by several different artists. The problem is that the statement lists all combinations of values from the two tables in which the artist ID values aren't the same. What you really need is a list of values in `artist` that aren't present

at all in painting, but an inner join can only produce results based on values that are present in both tables. It can't tell you anything about values that are missing from one of them.

When faced with the need to find values in one table with no match in (or missing from) another table, you should get in the habit of thinking, "Aha, that's a LEFT JOIN problem." A LEFT JOIN is one type of outer join: it's similar to an inner join in that it matches rows in the first (left) table with rows in the second (right) table. In addition, if a left table row has no match in the right table, a LEFT JOIN still produces a row—one in which all the columns from the right table are set to NULL. This means you can find values that are missing from the right table by looking for NULL. It's easier to understand how this happens by working in stages. Begin with an inner join that displays matching rows:

```
mysql> SELECT * FROM artist INNER JOIN painting
    -> ON artist.a_id = painting.a_id
    -> ORDER BY artist.a_id;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    1 |    2 | Mona Lisa         | MI    |    87 |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    4 | Renoir   |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+------+-------------------+-------+-------+
```

In this output, the first a_id column comes from the artist table and the second one comes from the painting table.

Now substitute LEFT for INNER to see the result you get from an outer join:

```
mysql> SELECT * FROM artist LEFT JOIN painting
    -> ON artist.a_id = painting.a_id
    -> ORDER BY artist.a_id;
+------+----------+------+------+-------------------+-------+-------+
| a_id | name     | a_id | p_id | title             | state | price |
+------+----------+------+------+-------------------+-------+-------+
|    1 | Da Vinci |    1 |    1 | The Last Supper   | IN    |    34 |
|    1 | Da Vinci |    1 |    2 | Mona Lisa         | MI    |    87 |
|    2 | Monet    | NULL | NULL | NULL              | NULL  |  NULL |
|    3 | Van Gogh |    3 |    3 | Starry Night      | KY    |    48 |
|    3 | Van Gogh |    3 |    4 | The Potato Eaters | KY    |    67 |
|    4 | Renoir   |    4 |    5 | Les Deux Soeurs   | NE    |    64 |
+------+----------+------+------+-------------------+-------+-------+
```

Compared to the inner join, the outer join produces an additional row for every artist row that has no painting table match, with all painting columns set to NULL.

Next, to restrict the output only to the unnmatched artist rows, add a WHERE clause that looks for NULL values in any painting column that cannot otherwise contain

NULL. This filters out the rows that the inner join produces, leaving those produced only by the outer join:

```
mysql> SELECT * FROM artist LEFT JOIN painting
    -> ON artist.a_id = painting.a_id
    -> WHERE painting.a_id IS NULL;
+------+-------+------+------+-------+-------+-------+
| a_id | name  | a_id | p_id | title | state | price |
+------+-------+------+------+-------+-------+-------+
|    2 | Monet | NULL | NULL | NULL  | NULL  |  NULL |
+------+-------+------+------+-------+-------+-------+
```

Finally, to show only the artist table values that are missing from the painting table, write the output column list to name only columns from the artist table. The result is that the LEFT JOIN lists those left-table rows containing a_id values not present in the right table:

```
mysql> SELECT artist.* FROM artist LEFT JOIN painting
    -> ON artist.a_id = painting.a_id
    -> WHERE painting.a_id IS NULL;
+------+-------+
| a_id | name  |
+------+-------+
|    2 | Monet |
+------+-------+
```

A similar kind of operation reports each left-table value along with an indicator as to whether it's present in the right table. To do this, perform a LEFT JOIN that counts the number of times each left-table value occurs in the right table. A count of zero indicates that the value is not present. The following statement lists each artist from the artist table and shows whether you have any paintings by the artist:

```
mysql> SELECT artist.name,
    -> IF(COUNT(painting.a_id)>0,'yes','no') AS 'in collection?'
    -> FROM artist LEFT JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
+----------+----------------+
| name     | in collection? |
+----------+----------------+
| Da Vinci | yes            |
| Monet    | no             |
| Renoir   | yes            |
| Van Gogh | yes            |
+----------+----------------+
```

A RIGHT JOIN is an outer join that is like LEFT JOIN but reverses the roles of the left and right tables. Semantically, RIGHT JOIN forces the matching process to produce a row from each table in the right table, even in the absence of a corresponding row in the left table. Syntactically, *tbl1* LEFT JOIN *tbl2* is equivalent to *tbl2* RIGHT JOIN *tbl1*. Therefore, references to LEFT JOIN in this book apply to RIGHT JOIN as well if you reverse the roles of the tables.

Another way to identify values present in one table but missing from another is to use a NOT IN subquery. The following example finds artists not represented in the painting table; compare it to the earlier LEFT JOIN that answers the same question:

```
mysql> SELECT * FROM artist
    -> WHERE a_id NOT IN (SELECT a_id FROM painting);
+------+-------+
| a_id | name  |
+------+-------+
|    2 | Monet |
+------+-------+
```

Other Ways to Write LEFT JOIN and RIGHT JOIN Queries

As with INNER JOIN, if the names of the columns to be matched in an outer join are the same in both tables and you compare them with the = operator, you can use a USING clause rather than ON. For example, the following two statements are equivalent:

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.n = t2.n;
SELECT * FROM t1 LEFT JOIN t2 USING (n);
```

As are these:

```
SELECT * FROM t1 RIGHT JOIN t2 ON t1.n = t2.n;
SELECT * FROM t1 RIGHT JOIN t2 USING (n);
```

In the special case that you want to base the comparison on every column that appears in both tables, you can use NATURAL LEFT JOIN or NATURAL RIGHT JOIN and omit the ON or USING clause:

```
SELECT * FROM t1 NATURAL LEFT JOIN t2;
SELECT * FROM t1 NATURAL RIGHT JOIN t2;
```

See Also

As shown in this section, LEFT JOIN is useful for finding values with no match in another table or for showing whether each value is matched. LEFT JOIN may also be used to produce a summary that includes all items in a list, even those for which there's nothing to summarize. This is very common for relationships between a master table and a detail table. For example, a LEFT JOIN can produce "total sales per customer" reports that list all customers, even those who bought nothing during the summary period. (For information about master-detail lists, see Recipe 14.5.)

LEFT JOIN is also useful for consistency checking when you receive two datafiles that are supposed to be related, and you want to determine whether they really are. (That is, you want to check the integrity of their relationship.) Import each file into a MySQL table, and then run a couple LEFT JOIN statements to determine whether there are unattached rows in one table or the other—that is, rows that have no match in the other

table. Recipe 14.3 discusses how to identify (and optionally delete) these unattached rows.

14.3. Identifying and Removing Mismatched or Unattached Rows

Problem

You have two datasets that are related, but possibly imperfectly so. You want to determine whether there are records in either dataset that are "unattached" (not matched by any record in the other dataset), and perhaps remove them if so.

Solution

To identify unmatched values in each table, use a `LEFT JOIN` or a `NOT IN` subquery. To remove them, use `DELETE` with a `NOT IN` subquery.

Discussion

Inner joins are useful for identifying matches, and outer joins are useful for identifying mismatches. This property of outer joins is valuable when you have related datasets for which the relationship might be imperfect. Mismatches might be found, for example, when you must verify the integrity of two datafiles received from an external source.

When you have related tables with unmatched rows, you can analyze and modify them using SQL statements. Specifically, restoring their relationship is a matter of identifying the unattached rows and then deleting them:

- To identify unattached rows, use a `LEFT JOIN`, because this is a "find unmatched rows" problem; alternatively, use a `NOT IN` subquery (see Recipe 14.2).
- To delete rows that are unmatched, use `DELETE` with a `NOT IN` subquery.

It's useful to know about unmatched data because you can alert whoever gave you the data. The data collection method might have a flaw that must be corrected. For example, with sales data, a missing region might mean that some regional manager didn't report in and the omission was overlooked.

The following example shows how to identify and remove mismatched rows using two datasets that describe sales regions and volume of sales per region. One dataset contains the ID and location of each region:

```
mysql> SELECT * FROM sales_region ORDER BY region_id;
+-----------+-----------------------+
| region_id | name                  |
+-----------+-----------------------+
```

```
|         1 | London, United Kingdom |
|         2 | Madrid, Spain          |
|         3 | Berlin, Germany        |
|         4 | Athens, Greece         |
+-----------+------------------------+
```

The other dataset contains sales volume figures. Each row contains the amount of sales for a given quarter of a year and indicates the sales region to which the row applies:

```
mysql> SELECT * FROM sales_volume ORDER BY region_id, year, quarter;
+-----------+------+---------+--------+
| region_id | year | quarter | volume |
+-----------+------+---------+--------+
|         1 | 2014 |       1 | 100400 |
|         1 | 2014 |       2 | 120000 |
|         3 | 2014 |       1 | 280000 |
|         3 | 2014 |       2 | 250000 |
|         5 | 2014 |       1 |  18000 |
|         5 | 2014 |       2 |  32000 |
+-----------+------+---------+--------+
```

A little visual inspection reveals that neither table is fully matched by the other. Sales regions 2 and 4 are not represented in the sales volume table, and the sales volume table contains rows for region 5, which is not in the sales region table. But we don't want to check the tables by inspection. We want to find unmatched rows by using SQL statements that do the work.

Mismatch identification is a matter of using outer joins. For example, to find sales regions for which there are no sales volume rows, use the following `LEFT JOIN`:

```
mysql> SELECT sales_region.region_id AS 'unmatched region row IDs'
    -> FROM sales_region LEFT JOIN sales_volume
    ->   ON sales_region.region_id = sales_volume.region_id
    -> WHERE sales_volume.region_id IS NULL;
+--------------------------+
| unmatched region row IDs |
+--------------------------+
|                        2 |
|                        4 |
+--------------------------+
```

Conversely, to find sales volume rows that are not associated with any known region, reverse the roles of the two tables:

```
mysql> SELECT sales_volume.region_id AS 'unmatched volume row IDs'
    -> FROM sales_volume LEFT JOIN sales_region
    ->   ON sales_volume.region_id = sales_region.region_id
    -> WHERE sales_region.region_id IS NULL;
+--------------------------+
| unmatched volume row IDs |
+--------------------------+
|                        5 |
```

```
|                         5 |
+---------------------------+
```

In this case, an ID appears more than once in the list if there are multiple volume rows for a missing region. To see each unmatched ID only once, use `SELECT DISTINCT`:

```
mysql> SELECT DISTINCT sales_volume.region_id AS 'unmatched volume row IDs'
    -> FROM sales_volume LEFT JOIN sales_region
    ->   ON sales_volume.region_id = sales_region.region_id
    -> WHERE sales_region.region_id IS NULL
+---------------------------+
| unmatched volume row IDs |
+---------------------------+
|                         5 |
+---------------------------+
```

You can also identify mismatches using `NOT IN` subqueries:

```
mysql> SELECT region_id AS 'unmatched region row IDs'
    -> FROM sales_region
    -> WHERE region_id NOT IN (SELECT region_id FROM sales_volume);
+---------------------------+
| unmatched region row IDs |
+---------------------------+
|                         2 |
|                         4 |
+---------------------------+
mysql> SELECT region_id AS 'unmatched volume row IDs'
    -> FROM sales_volume
    -> WHERE region_id NOT IN (SELECT region_id FROM sales_region);
+---------------------------+
| unmatched volume row IDs |
+---------------------------+
|                         5 |
|                         5 |
+---------------------------+
```

To get rid of unmatched rows, use a `NOT IN` subquery in a `DELETE` statement. To remove `sales_region` rows that match no `sales_volume` rows, do this:

```
DELETE FROM sales_region
WHERE region_id NOT IN (SELECT region_id FROM sales_volume);
```

To remove mismatched `sales_volume` rows that match no `sales_region` rows, the statement is similar but with the table roles reversed:

```
DELETE FROM sales_volume
WHERE region_id NOT IN (SELECT region_id FROM sales_region);
```

Using Foreign Keys to Enforce Referential Integrity and Prevent Mismatches

One feature a database system offers to help you maintain consistency between tables is the ability to define foreign key relationships. This means you can specify explicitly in the table definition that a primary key in a parent table (such as the `region_id` column of the `sales_region` table) is a parent to a key in another table (the `region_id` column in the `sales_volume` table).

By defining the ID column in the child table as a foreign key to the ID column in the parent, the database system can enforce certain constraints against illegal operations. For example, it can prevent you from creating a child row with an ID not present in the parent or from deleting parent rows without also deleting the corresponding child rows first. A foreign key implementation may also offer cascaded delete and update: if you delete or update a parent row, the database engine cascades the effect of the delete or update to any child tables and automatically deletes or updates the child rows for you. The InnoDB storage engine in MySQL supports foreign keys and cascaded deletes and updates.

14.4. Comparing a Table to Itself

Problem

You want to compare rows in a table to other rows in the same table. For example, you want to find all paintings in your collection by the artist who painted *The Potato Eaters*. Or you want to know which states listed in the `states` table joined the Union in the same year as New York. Or you want to know which states did not join the Union in the same year as any other state.

Solution

Problems that require comparing a table to itself involve an operation known as a self-join. It's performed much like other joins, except that you must use table aliases so that you can refer to the same table different ways within the statement.

Discussion

A special case of joining one table to another occurs when both tables are the same. This is called a self-join. This may be confusing or strange to think about at first, but it's perfectly legal. You'll likely find yourself using self-joins quite often because they are so important.

A tip-off that a self-join is required is that you want to know which pairs of rows in a table satisfy some condition. Suppose that your favorite painting is *The Potato Eaters*, and you want to identify all items in your collection that were painted by the same artist. The artist ID and painting titles that we begin with look like this:

```
mysql> SELECT a_id, title FROM painting ORDER BY a_id;
+------+------------------+
| a_id | title            |
+------+------------------+
|    1 | The Last Supper  |
|    1 | Mona Lisa        |
|    3 | Starry Night     |
|    3 | The Potato Eaters |
|    4 | Les Deux Soeurs  |
+------+------------------+
```

Solve the problem as follows:

1. Identify which `painting` table row contains the title *The Potato Eaters*, so that you can refer to its `a_id` value.
2. Match other rows in the table that have the same `a_id` value.
3. Display the titles from those matching rows.

The trick lies in using the proper notation. First attempts at joining a table to itself often look something like this:

```
mysql> SELECT title
    -> FROM painting INNER JOIN painting
    -> ON a_id = a_id;
    -> WHERE title = 'The Potato Eaters';
ERROR 1066 (42000): Not unique table/alias: 'painting'
```

The column references in that statement are ambiguous because MySQL cannot tell which instance of the `painting` table any given column name refers to. The solution is to alias at least one instance of the table so that you can distinguish column references by using different table qualifiers. The following statement shows how to do this, using the aliases `p1` and `p2` to refer to the `painting` table different ways:

```
mysql> SELECT p2.title
    -> FROM painting AS p1 INNER JOIN painting AS p2
    -> ON p1.a_id = p2.a_id
    -> WHERE p1.title = 'The Potato Eaters';
+------------------+
| title            |
+------------------+
| Starry Night     |
| The Potato Eaters |
+------------------+
```

The statement output illustrates something typical of self-joins: when you begin with a reference value in one table instance (*The Potato Eaters*) to find matching rows in a second table instance (paintings by the same artist), the output includes the reference value. That makes sense: after all, the reference matches itself. To find only *other* paintings by the same artist, explicitly exclude the reference value from the output:

```
mysql> SELECT p2.title
    -> FROM painting AS p1 INNER JOIN painting AS p2
    -> ON p1.a_id = p2.a_id
    -> WHERE p1.title = 'The Potato Eaters' AND p2.title <> p1.title
+--------------+
| title        |
+--------------+
| Starry Night |
+--------------+
```

The preceding statements use ID value comparisons to match rows in the two table instances, but any kind of value can be used. For example, to use the `states` table to answer the question "Which states joined the Union in the same year as New York?" perform a temporal pairwise comparison based on the year part of the dates in the table's `statehood` column:

```
mysql> SELECT s2.name, s2.statehood
    -> FROM states AS s1 INNER JOIN states AS s2
    -> ON YEAR(s1.statehood) = YEAR(s2.statehood) AND s1.name <> s2.name
    -> WHERE s1.name = 'New York'
    -> ORDER BY s2.name;
+----------------+------------+
| name           | statehood  |
+----------------+------------+
| Connecticut    | 1788-01-09 |
| Georgia        | 1788-01-02 |
| Maryland       | 1788-04-28 |
| Massachusetts  | 1788-02-06 |
| New Hampshire  | 1788-06-21 |
| South Carolina | 1788-05-23 |
| Virginia       | 1788-06-25 |
+----------------+------------+
```

Now suppose that you want to find *every* pair of states that joined the Union in the same year. In this case, the output potentially can include any pair of rows from the `states` table. A self-join is perfect for this problem:

```
mysql> SELECT YEAR(s1.statehood) AS year,
    -> s1.name AS name1, s1.statehood AS statehood1,
    -> s2.name AS name2, s2.statehood AS statehood2
    -> FROM states AS s1 INNER JOIN states AS s2
    -> ON YEAR(s1.statehood) = YEAR(s2.statehood) AND s1.name <> s2.name
    -> ORDER BY year, name1, name2;
+------+---------------+------------+---------------+------------+
| year | name1         | statehood1 | name2         | statehood2 |
```

```
+------+---------------+------------+---------------+------------+
| 1787 | Delaware      | 1787-12-07 | New Jersey    | 1787-12-18 |
| 1787 | Delaware      | 1787-12-07 | Pennsylvania  | 1787-12-12 |
| 1787 | New Jersey    | 1787-12-18 | Delaware      | 1787-12-07 |
| 1787 | New Jersey    | 1787-12-18 | Pennsylvania  | 1787-12-12 |
| 1787 | Pennsylvania  | 1787-12-12 | Delaware      | 1787-12-07 |
| 1787 | Pennsylvania  | 1787-12-12 | New Jersey    | 1787-12-18 |
…
| 1912 | Arizona       | 1912-02-14 | New Mexico    | 1912-01-06 |
| 1912 | New Mexico    | 1912-01-06 | Arizona       | 1912-02-14 |
| 1959 | Alaska        | 1959-01-03 | Hawaii        | 1959-08-21 |
| 1959 | Hawaii        | 1959-08-21 | Alaska        | 1959-01-03 |
+------+---------------+------------+---------------+------------+
```

The condition in the `ON` clause that requires state pair names not to be identical eliminates the trivially duplicate rows showing that each state joined the Union in the same year as itself. But you'll notice that each remaining pair of states still appears twice. For example, there is one row that lists Delaware and New Jersey, and another that lists New Jersey and Delaware. This is often the case with self-joins: they produce pairs of rows that contain the same values, but for which the values are not in the same order.

Because the values are not listed in the same order within the rows, they are not identical and you can't get rid of these "near duplicates" by adding `DISTINCT` to the statement. To solve this problem, select rows in such a way that only one row from each pair ever appears in the query result. Slightly modify the `ON` clause, from:

```
ON YEAR(s1.statehood) = YEAR(s2.statehood) AND s1.name <> s2.name
```

to:

```
ON YEAR(s1.statehood) = YEAR(s2.statehood) AND s1.name < s2.name
```

Using `<` rather than `<>` selects only those rows in which the first state name is lexically less than the second, and eliminates rows in which the names appear in opposite order (as well as rows in which the state names are identical). The resulting query produces the desired output without duplicates:

```
mysql> SELECT YEAR(s1.statehood) AS year,
    -> s1.name AS name1, s1.statehood AS statehood1,
    -> s2.name AS name2, s2.statehood AS statehood2
    -> FROM states AS s1 INNER JOIN states AS s2
    -> ON YEAR(s1.statehood) = YEAR(s2.statehood) AND s1.name < s2.name
    -> ORDER BY year, name1, name2;
+------+---------------+------------+---------------+------------+
| year | name1         | statehood1 | name2         | statehood2 |
+------+---------------+------------+---------------+------------+
| 1787 | Delaware      | 1787-12-07 | New Jersey    | 1787-12-18 |
| 1787 | Delaware      | 1787-12-07 | Pennsylvania  | 1787-12-12 |
| 1787 | New Jersey    | 1787-12-18 | Pennsylvania  | 1787-12-12 |
…
| 1912 | Arizona       | 1912-02-14 | New Mexico    | 1912-01-06 |
```

```
| 1959 | Alaska         | 1959-01-03 | Hawaii         | 1959-08-21 |
+------+----------------+------------+----------------+------------+
```

For self-join problems of the "Which values are *not* matched by other rows in the table?" variety, use a `LEFT JOIN` rather than an `INNER JOIN`. An instance of this is the question "Which states did not join the Union in the same year as any other state?" In this case, the solution uses a `LEFT JOIN` of the `states` table to itself:

```
mysql> SELECT s1.name, s1.statehood
    -> FROM states AS s1 LEFT JOIN states AS s2
    -> ON YEAR(s1.statehood) = YEAR(s2.statehood) AND s1.name <> s2.name
    -> WHERE s2.name IS NULL
    -> ORDER BY s1.name;
+----------------+------------+
| name           | statehood  |
+----------------+------------+
| Alabama        | 1819-12-14 |
| Arkansas       | 1836-06-15 |
| California     | 1850-09-09 |
| Colorado       | 1876-08-01 |
| Illinois       | 1818-12-03 |
| Indiana        | 1816-12-11 |
| Iowa           | 1846-12-28 |
| Kansas         | 1861-01-29 |
| Kentucky       | 1792-06-01 |
…
| Tennessee      | 1796-06-01 |
| Utah           | 1896-01-04 |
| Vermont        | 1791-03-04 |
| West Virginia  | 1863-06-20 |
| Wisconsin      | 1848-05-29 |
+----------------+------------+
```

For each row in the `states` table, the statement selects rows for which the state has a `statehood` value in the same year, not including that state itself. For rows having no such match, the `LEFT JOIN` forces the output to contain a row anyway, with all the `s2` columns set to `NULL`. Those rows identify the states with no other state that joined the Union in the same year.

14.5. Producing Master-Detail Lists and Summaries

Problem

Two tables have a master-detail relationship, and you want to produce a list that shows each master row with its detail rows or a list that produces a summary of the detail rows for each master row.

Solution

This is a one-to-many relationship. The solution to this problem involves a join, but the type of join depends on the question you want answered. To produce a list containing only master rows for which some detail row exists, use an inner join based on the primary key in the master table. To produce a list that includes all master rows, even those with no detail rows, use an outer join.

Discussion

To produce a list from two tables that have a master-detail or parent-child relationship, a given row in one table might be matched by several rows in the other. These relationships occur frequently. For example, in business contexts, one-to-many relationships involve invoices per customer or items per invoice.

This section suggests some master-detail questions that you can ask (and answer) using the `artist` and `painting` tables from earlier in the chapter.

One form of master-detail question for these tables is, "Which paintings did each artist paint?" This is a simple inner join (see Recipe 14.1). Match each `artist` row to its corresponding `painting` rows based on the artist ID values:

```
mysql> SELECT artist.name, painting.title
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> ORDER BY name, title;
+----------+------------------+
| name     | title            |
+----------+------------------+
| Da Vinci | Mona Lisa        |
| Da Vinci | The Last Supper  |
| Renoir   | Les Deux Soeurs  |
| Van Gogh | Starry Night     |
| Van Gogh | The Potato Eaters |
+----------+------------------+
```

To also list artists for whom you have no paintings, the join output should include rows in one table that have no match in the other. That's a form of "find the nonmatching rows" problem that requires an outer join (see Recipe 14.2). Thus, to list each `artist` row, whether or not any `painting` rows match, use a `LEFT JOIN`:

```
mysql> SELECT artist.name, painting.title
    -> FROM artist LEFT JOIN painting ON artist.a_id = painting.a_id
    -> ORDER BY name, title;
+----------+------------------+
| name     | title            |
+----------+------------------+
| Da Vinci | Mona Lisa        |
| Da Vinci | The Last Supper  |
| Monet    | NULL             |
| Renoir   | Les Deux Soeurs  |
```

```
| Van Gogh | Starry Night      |
| Van Gogh | The Potato Eaters |
+----------+-------------------+
```

Rows in the result that have `NULL` in the `title` column correspond to artists listed in the `artist` table for whom you have no paintings.

The same principles apply when producing summaries using master and detail tables. For example, to summarize your art collection by number of paintings per artist, you might ask, "How many paintings are there per artist in the `painting` table?" To find the answer based on artist ID but display artist name (from the `artist` table), count the paintings with this statement:

```
mysql> SELECT artist.name, COUNT(painting.a_id) AS paintings
    -> FROM artist INNER JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
+----------+-----------+
| name     | paintings |
+----------+-----------+
| Da Vinci |         2 |
| Renoir   |         1 |
| Van Gogh |         2 |
+----------+-----------+
```

On the other hand, you might ask, "How many paintings did each artist paint?" This is the same question as the previous one (and the same statement answers it), as long as every artist in the `artist` table has at least one corresponding `painting` table row. But if you have artists in the `artist` table not yet represented by any paintings in your collection, they do not appear in the statement output. To produce a summary that also includes artists with no paintings in the `painting` table, use a `LEFT JOIN`:

```
mysql> SELECT artist.name, COUNT(painting.a_id) AS paintings
    -> FROM artist LEFT JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
+----------+-----------+
| name     | paintings |
+----------+-----------+
| Da Vinci |         2 |
| Monet    |         0 |
| Renoir   |         1 |
| Van Gogh |         2 |
+----------+-----------+
```

Beware of a subtle error that is easy to make when writing that kind of statement. Suppose that you write the `COUNT()` function slightly differently, like so:

```
mysql> SELECT artist.name, COUNT(*) AS paintings
    -> FROM artist LEFT JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
```

```
+----------+-----------+
| name     | paintings |
+----------+-----------+
| Da Vinci |         2 |
| Monet    |         1 |
| Renoir   |         1 |
| Van Gogh |         2 |
+----------+-----------+
```

Now every artist appears to have at least one painting. Why the difference? The problem is the use of `COUNT(*)` rather than `COUNT(painting.a_id)`. The way `LEFT JOIN` works for unmatched rows in the left table is that it generates a row with all the columns from the right table set to `NULL`. In the example, the right table is `painting`. The statement that uses `COUNT(painting.a_id)` works correctly because `COUNT(`*`expr`*`)` counts only non-`NULL` values. The statement that uses `COUNT(*)` is incorrect because it counts *rows*, including those containing `NULL` that correspond to missing artists.

`LEFT JOIN` is suitable for other types of summaries as well. To produce additional columns showing the total and average prices of the paintings for each artist in the `artist` table, use this statement:

```
mysql> SELECT artist.name,
    -> COUNT(painting.a_id) AS 'number of paintings',
    -> SUM(painting.price) AS 'total price',
    -> AVG(painting.price) AS 'average price'
    -> FROM artist LEFT JOIN painting ON artist.a_id = painting.a_id
    -> GROUP BY artist.name;
+----------+--------------------+-------------+---------------+
| name     | number of paintings | total price | average price |
+----------+--------------------+-------------+---------------+
| Da Vinci |                  2 |         121 |       60.5000 |
| Monet    |                  0 |        NULL |          NULL |
| Renoir   |                  1 |          64 |       64.0000 |
| Van Gogh |                  2 |         115 |       57.5000 |
+----------+--------------------+-------------+---------------+
```

Note that `COUNT()` is zero for artists that are not represented, but `SUM()` and `AVG()` are `NULL`. The latter two functions return `NULL` when applied to a set of values with no non-`NULL` values. To display a sum or average value of zero in that case, replace `SUM(`*`expr`*`)` and `AVG(`*`expr`*`)` with `IFNULL(SUM(`*`expr`*`),0)` and `IFNULL(AVG(`*`expr`*`),0)`.

14.6. Enumerating a Many-to-Many Relationship

Problem

You want to display a relationship between tables when any row in either table might be matched by multiple rows in the other.

Solution

This is a many-to-many relationship. It requires a third table for associating your two primary tables and a three-way join to produce the correspondences between them.

Discussion

The `artist` and `painting` tables used in earlier sections have a one-to-many relationship: a given artist may have produced many paintings, but each painting was created by only one artist. One-to-many relationships are relatively simple and the two related tables can be joined using a column that is common to both.

A many-to-many relationship between tables is more complex. It occurs when a row in one table may have many matches in the other, and vice versa. An example is the relationship between movies and actors: each movie may have multiple actors, and each actor may have appeared in multiple movies. One way to represent this relationship uses a table structured as follows, with a row for each movie-actor combination:

```
mysql> SELECT * FROM movies_actors ORDER BY year, movie, actor;
+------+---------------------------+---------------+
| year | movie                     | actor         |
+------+---------------------------+---------------+
| 1997 | The Fifth Element         | Bruce Willis  |
| 1997 | The Fifth Element         | Gary Oldman   |
| 1997 | The Fifth Element         | Ian Holm      |
| 1999 | The Phantom Menace        | Ewan McGregor |
| 1999 | The Phantom Menace        | Liam Neeson   |
| 2001 | The Fellowship of the Ring | Elijah Wood   |
| 2001 | The Fellowship of the Ring | Ian Holm      |
| 2001 | The Fellowship of the Ring | Ian McKellen  |
| 2001 | The Fellowship of the Ring | Orlando Bloom |
| 2005 | Kingdom of Heaven         | Liam Neeson   |
| 2005 | Kingdom of Heaven         | Orlando Bloom |
| 2010 | Red                       | Bruce Willis  |
| 2010 | Red                       | Helen Mirren  |
| 2011 | Unknown                   | Diane Kruger  |
| 2011 | Unknown                   | Liam Neeson   |
+------+---------------------------+---------------+
```

The table captures the nature of this many-to-many relationship, but it's also in non-normal form because it unnecessarily stores repetitive information. For example, information for each movie is recorded multiple times. To better represent this many-to-many relationship, use multiple tables:

- Store each movie year and name once in a table named `movies`.
- Store each actor name once in a table named `actors`.
- Create a third table, `movies_actors_link`, that stores movie-actor associations and serves as a link, or bridge, between the two primary tables. To minimize the infor-

mation stored in this table, assign unique IDs to each movie and actor within their respective tables, and store only those IDs in the `movies_actors_link` table.

The resulting `movie` and `actor` tables look like this:

```
mysql> SELECT * FROM movies ORDER BY id;
+----+------+---------------------------+
| id | year | movie                     |
+----+------+---------------------------+
|  1 | 1997 | The Fifth Element         |
|  2 | 1999 | The Phantom Menace        |
|  3 | 2001 | The Fellowship of the Ring |
|  4 | 2005 | Kingdom of Heaven         |
|  5 | 2010 | Red                       |
|  6 | 2011 | Unknown                   |
+----+------+---------------------------+
mysql> SELECT * FROM actors ORDER BY id;
+----+---------------+
| id | actor         |
+----+---------------+
|  1 | Bruce Willis  |
|  2 | Diane Kruger  |
|  3 | Elijah Wood   |
|  4 | Ewan McGregor |
|  5 | Gary Oldman   |
|  6 | Helen Mirren  |
|  7 | Ian Holm      |
|  8 | Ian McKellen  |
|  9 | Liam Neeson   |
| 10 | Orlando Bloom |
+----+---------------+
```

The `movies_actors_link` table associates movies and actors as follows:

```
mysql> SELECT * FROM movies_actors_link ORDER BY movie_id, actor_id;
+----------+----------+
| movie_id | actor_id |
+----------+----------+
|        1 |        1 |
|        1 |        5 |
|        1 |        7 |
|        2 |        4 |
|        2 |        9 |
|        3 |        3 |
|        3 |        7 |
|        3 |        8 |
|        3 |       10 |
|        4 |        9 |
|        4 |       10 |
|        5 |        1 |
|        5 |        6 |
|        6 |        2 |
```

```
|       6 |       9 |
+---------+---------+
```

You'll surely notice that the content of the `movies_actors_link` table is entirely meaningless from a human perspective. That's okay: we need never display it explicitly. Its utility derives from its ability to link the two primary tables in queries, without appearing in query output itself. The next few examples illustrate this principle. They answer questions about the movies or actors, using three-way joins that relate the two primary tables using the link table.

- List all the pairings that show each movie and who acted in it. This statement enumerates all the correspondences between the `movie` and `actor` tables and reproduces the information that was originally in the nonnormal `movies_actors` table:

```
mysql> SELECT m.year, m.movie, a.actor
    -> FROM movies AS m INNER JOIN movies_actors_link AS l
    -> INNER JOIN actors AS a
    -> ON m.id = l.movie_id AND a.id = l.actor_id
    -> ORDER BY m.year, m.movie, a.actor;
+------+---------------------------+---------------+
| year | movie                     | actor         |
+------+---------------------------+---------------+
| 1997 | The Fifth Element         | Bruce Willis  |
| 1997 | The Fifth Element         | Gary Oldman   |
| 1997 | The Fifth Element         | Ian Holm      |
| 1999 | The Phantom Menace        | Ewan McGregor |
| 1999 | The Phantom Menace        | Liam Neeson   |
| 2001 | The Fellowship of the Ring | Elijah Wood   |
| 2001 | The Fellowship of the Ring | Ian Holm      |
| 2001 | The Fellowship of the Ring | Ian McKellen  |
| 2001 | The Fellowship of the Ring | Orlando Bloom |
| 2005 | Kingdom of Heaven         | Liam Neeson   |
| 2005 | Kingdom of Heaven         | Orlando Bloom |
| 2010 | Red                       | Bruce Willis  |
| 2010 | Red                       | Helen Mirren  |
| 2011 | Unknown                   | Diane Kruger  |
| 2011 | Unknown                   | Liam Neeson   |
+------+---------------------------+---------------+
```

- List the actors in a given movie:

```
mysql> SELECT a.actor
    -> FROM movies AS m INNER JOIN movies_actors_link AS l
    -> INNER JOIN actors AS a
    -> ON m.id = l.movie_id AND a.id = l.actor_id
    -> WHERE m.movie = 'The Fellowship of the Ring'
    -> ORDER BY a.actor;
+---------------+
| actor         |
+---------------+
| Elijah Wood   |
| Ian Holm      |
```

```
| Ian McKellen  |
| Orlando Bloom |
+---------------+
```

- List the movies in which a given actor has acted:

```
mysql> SELECT m.year, m.movie
    -> FROM movies AS m INNER JOIN movies_actors_link AS l
    -> INNER JOIN actors AS a
    -> ON m.id = l.movie_id AND a.id = l.actor_id
    -> WHERE a.actor = 'Liam Neeson'
    -> ORDER BY m.year, m.movie;
+------+--------------------+
| year | movie              |
+------+--------------------+
| 1999 | The Phantom Menace |
| 2005 | Kingdom of Heaven  |
| 2011 | Unknown            |
+------+--------------------+
```

14.7. Finding Per-Group Minimum or Maximum Values

Problem

You want to find which row within each group of rows in a table contains the maximum or minimum value for a given column. For example, you want to determine the most expensive painting in your collection for each artist.

Solution

Create a temporary table to hold the per-group maximum or minimum values, then join the temporary table with the original one to pull out the matching row for each group. If you prefer a single-query solution, use a subquery in the `FROM` clause rather than a temporary table.

Discussion

Many questions involve finding largest or smallest values in a particular table column, but it's also common to want to know other values in the row that contains the value. For example, using the `artist` and `painting` tables with the techniques from Recipe 8.3, it's possible to answer questions such as "What is the most expensive painting in the collection, and who painted it?" One solution is to store the highest price in a user-defined variable, then use the variable to identify the row containing the price so that you can retrieve other columns from it:

```
mysql> SET @max_price = (SELECT MAX(price) FROM painting);
mysql> SELECT artist.name, painting.title, painting.price
```

```
    -> FROM artist INNER JOIN painting
    -> ON painting.a_id = artist.a_id
    -> WHERE painting.price = @max_price;
+----------+-----------+-------+
| name     | title     | price |
+----------+-----------+-------+
| Da Vinci | Mona Lisa |    87 |
+----------+-----------+-------+
```

The same thing can be done by creating a temporary table to hold the maximum price and joining it with the other tables:

```
CREATE TABLE tmp SELECT MAX(price) AS max_price FROM painting;
SELECT artist.name, painting.title, painting.price
FROM artist INNER JOIN painting INNER JOIN tmp
ON painting.a_id = artist.a_id
AND painting.price = tmp.max_price;
```

On the face of it, using a temporary table and a join is just a more complicated way of answering the question than with a user-defined variable. Does this technique have any practical value? Yes, it does, because it leads to a more general technique for answering more difficult questions. The previous statements show information only for the single most expensive painting in the entire `painting` table. What if your question is, "What is the most expensive painting *for each artist*?" You can't use a user-defined variable to answer that question because the answer requires finding one price per artist, and a variable holds only a single value. But the technique of using a temporary table works well because the table can hold multiple rows, and a join can find matches for all of them.

To answer the question, select each artist ID and the corresponding maximum painting price into a temporary table. This table contains not only the maximum painting price but the maximum within each group, where "group" is defined as "paintings by a given artist." Then use the artist IDs and prices stored in the temporary table to match rows in the `painting` table, and join the result with the `artist` table to get the artist names:

```
mysql> CREATE TABLE tmp
    -> SELECT a_id, MAX(price) AS max_price FROM painting GROUP BY a_id;
mysql> SELECT artist.name, painting.title, painting.price
    -> FROM artist INNER JOIN painting INNER JOIN tmp
    -> ON painting.a_id = artist.a_id
    -> AND painting.a_id = tmp.a_id
    -> AND painting.price = tmp.max_price;
+----------+------------------+-------+
| name     | title            | price |
+----------+------------------+-------+
| Da Vinci | Mona Lisa        |    87 |
| Van Gogh | The Potato Eaters |    67 |
| Renoir   | Les Deux Soeurs  |    64 |
+----------+------------------+-------+
```

To obtain the same result with a single statement, use a subquery in the FROM clause that retrieves the same rows contained in the temporary table:

```
SELECT artist.name, painting.title, painting.price
FROM artist INNER JOIN painting INNER JOIN
(SELECT a_id, MAX(price) AS max_price FROM painting GROUP BY a_id) AS tmp
ON painting.a_id = artist.a_id
AND painting.a_id = tmp.a_id
AND painting.price = tmp.max_price;
```

Yet another way to answer maximum-per-group questions is to use a LEFT JOIN that joins a table to itself. The following statement identifies the highest-priced painting per artist ID (use IS NULL to select all the rows from p1 for which there is *no* row in p2 with a higher price):

```
mysql> SELECT p1.a_id, p1.title, p1.price
    -> FROM painting AS p1 LEFT JOIN painting AS p2
    -> ON p1.a_id = p2.a_id AND p1.price < p2.price
    -> WHERE p2.a_id IS NULL;
+------+------------------+-------+
| a_id | title            | price |
+------+------------------+-------+
|    1 | Mona Lisa        |    87 |
|    3 | The Potato Eaters |    67 |
|    4 | Les Deux Soeurs  |    64 |
+------+------------------+-------+
```

To display artist names rather than ID values, join the result of the LEFT JOIN to the artist table:

```
mysql> SELECT artist.name, p1.title, p1.price
    -> FROM painting AS p1 LEFT JOIN painting AS p2
    -> ON p1.a_id = p2.a_id AND p1.price < p2.price
    -> INNER JOIN artist ON p1.a_id = artist.a_id
    -> WHERE p2.a_id IS NULL;
+----------+------------------+-------+
| name     | title            | price |
+----------+------------------+-------+
| Da Vinci | Mona Lisa        |    87 |
| Van Gogh | The Potato Eaters |    67 |
| Renoir   | Les Deux Soeurs  |    64 |
+----------+------------------+-------+
```

The self-LEFT JOIN method is perhaps less intuitive than using a temporary table or a subquery.

Which technique is better: the temporary table or the subquery in the FROM clause? For small tables, there might not be much difference either way. If the temporary table or subquery result is large, a general advantage of the temporary table is that you can index it after creating it and before using it in a join. However, as of MySQL 5.6, the optimizer automatically adds an index to subquery results in the FROM clause if it estimates that

will speed up query execution. Thus, the disadvantage of the subquery goes away and you can use it freely without concern over whether to use the temporary table instead.

See Also

This section shows how to answer maximum-per-group questions by selecting summary information into a temporary table and joining that table to the original one or by using a subquery in the `FROM` clause. These techniques have application in many contexts. One of them is calculation of team standings, where the standings for each group of teams are determined by comparing each team in the group to the team with the best record. Recipe 15.12 discusses how to do this.

14.8. Using a Join to Fill or Identify Holes in a List

Problem

You want to produce a summary by category, but some categories are missing from the data to be summarized. Consequently, the summary has missing categories as well.

Solution

Create a reference table that lists each category and produce the summary based on a `LEFT JOIN` between the list and the table containing your data. Every category in the reference table will appear in the result, even those not present in the data to be summarized.

Discussion

A summary query normally produces entries only for categories actually present in the data. Suppose that you want to summarize the `driver_log` table (introduced in Chapter 7), to determine how many drivers were on the road each day. The table has these rows:

```
mysql> SELECT * FROM driver_log ORDER BY rec_id;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      1 | Ben   | 2014-07-30 |   152 |
|      2 | Suzi  | 2014-07-29 |   391 |
|      3 | Henry | 2014-07-29 |   300 |
|      4 | Henry | 2014-07-27 |    96 |
|      5 | Ben   | 2014-07-29 |   131 |
|      6 | Henry | 2014-07-26 |   115 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      8 | Henry | 2014-08-01 |   197 |
|      9 | Ben   | 2014-08-02 |    79 |
```

```
|     10 | Henry | 2014-07-30 |   203 |
+--------+-------+------------+-------+
```

A simple summary showing the number of active drivers per day looks like this:

```
mysql> SELECT trav_date, COUNT(trav_date) AS drivers
    -> FROM driver_log GROUP BY trav_date ORDER BY trav_date;
+------------+---------+
| trav_date  | drivers |
+------------+---------+
| 2014-07-26 |       1 |
| 2014-07-27 |       1 |
| 2014-07-29 |       3 |
| 2014-07-30 |       2 |
| 2014-08-01 |       1 |
| 2014-08-02 |       2 |
+------------+---------+
```

Here, the summary category is date, but the summary is "incomplete" in the sense that it includes entries only for dates represented in the `driver_log` table. To produce a summary that includes all categories (all dates within the date range represented in the table), including those for which no driver was active, create a reference table that lists each date:

```
mysql> CREATE TABLE dates (d DATE);
mysql> INSERT INTO dates (d)
    -> VALUES('2014-07-26'),('2014-07-27'),('2014-07-28'),
    -> ('2014-07-29'),('2014-07-30'),('2014-07-31'),
    -> ('2014-08-01'),('2014-08-02');
```

Then join the reference table to the `driver_log` table using a `LEFT JOIN`:

```
mysql> SELECT dates.d, COUNT(driver_log.trav_date) AS drivers
    -> FROM dates LEFT JOIN driver_log ON dates.d = driver_log.trav_date
    -> GROUP BY d ORDER BY d;
+------------+---------+
| d          | drivers |
+------------+---------+
| 2014-07-26 |       1 |
| 2014-07-27 |       1 |
| 2014-07-28 |       0 |
| 2014-07-29 |       3 |
| 2014-07-30 |       2 |
| 2014-07-31 |       0 |
| 2014-08-01 |       1 |
| 2014-08-02 |       2 |
+------------+---------+
```

Now the summary includes a row for every date in the range because the `LEFT JOIN` forces the output to include a row for every date in the reference table, even those missing from the `driver_log` table.

The example just shown uses the reference table with a `LEFT JOIN` to fill holes in the summary. It's also possible to use the reference table to *detect* holes in the dataset—that is, to determine which categories are not present in the data to be summarized. The following statement shows those dates on which no driver was active by looking for reference rows for which no `driver_log` table rows have a matching category value:

```
mysql> SELECT dates.d
    -> FROM dates LEFT JOIN driver_log ON dates.d = driver_log.trav_date
    -> WHERE driver_log.trav_date IS NULL;
+------------+
| d          |
+------------+
| 2014-07-28 |
| 2014-07-31 |
+------------+
```

Reference tables that contain a list of categories are quite useful in summary context, as just shown. But creating such tables manually is mind-numbing and error-prone. A stored procedure that uses the endpoints of the range of category values to generate the reference table for you helps automate the process. In essence, this type of procedure acts as an iterator that generates a row for each value in the range. The following procedure, `make_date_list()`, shows an example of this approach. It creates a reference table containing a row for every date in a particular date range. It also indexes the table so that it will be fast in large joins:

```
CREATE PROCEDURE make_date_list(db_name TEXT, tbl_name TEXT, col_name TEXT,
                                min_date DATE, max_date DATE)
BEGIN
  DECLARE i, days INT;
  SET i = 0, days = DATEDIFF(max_date,min_date)+1;

  # Make identifiers safe for insertion into SQL statements. Use db_name
  # and tbl_name to create qualified table name.
  SET tbl_name = CONCAT(quote_identifier(db_name),'.',
                        quote_identifier(tbl_name));
  SET col_name = quote_identifier(col_name);
  CALL exec_stmt(CONCAT('DROP TABLE IF EXISTS ',tbl_name));
  CALL exec_stmt(CONCAT('CREATE TABLE ',tbl_name,'(',
                        col_name,' DATE NOT NULL, PRIMARY KEY(',
                        col_name,'))'));
  WHILE i < days DO
    CALL exec_stmt(CONCAT('INSERT INTO ',tbl_name,'(',col_name,') VALUES(',
                          QUOTE(min_date),' + INTERVAL ',i,' DAY)'));
    SET i = i + 1;
  END WHILE;
END;
```

Use `make_date_list()` to generate the reference table, `dates`, like this:

```
CALL make_date_list('cookbook', 'dates', 'd', '2014-07-26', '2014-08-02');
```

Then use the `dates` table as shown earlier in this section to fill holes in the summary or to detect holes in the dataset.

You can find the `make_date_list()` procedure in the *joins* directory of the `recipes` distribution. It requires the `exec_stmt()` and `quote_identifier()` helper routines (see Recipe 9.9), located in the *routines* directory. The *joins* directory also contains a Perl script, *make_date_list.pl*, that implements an alternate approach; it generates date reference tables from the command line.

14.9. Using a Join to Control Query Sort Order

Problem

You want to sort a statement's output using a characteristic of the output that cannot be specified using `ORDER BY`. For example, you want to sort a set of rows by subgroups, putting first those groups with the most rows and last those groups with the fewest rows. But "number of rows in each group" is not a property of individual rows, so you can't use it for sorting.

Solution

Derive the ordering information and store it in an auxiliary table. Then join the original table to the auxiliary table, using the auxiliary table to control the sort order.

Discussion

Most of the time you sort a query result using an `ORDER BY` clause that names which column or columns to use for sorting. But sometimes the values you want to sort by aren't present in the rows to be sorted. This is the case when you want to use group characteristics to order the rows. The following example uses the `driver_log` table to illustrate this. The following query sorts the table using the ID column, which is present in the rows:

```
mysql> SELECT * FROM driver_log ORDER BY rec_id;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      1 | Ben   | 2014-07-30 |   152 |
|      2 | Suzi  | 2014-07-29 |   391 |
|      3 | Henry | 2014-07-29 |   300 |
|      4 | Henry | 2014-07-27 |    96 |
|      5 | Ben   | 2014-07-29 |   131 |
|      6 | Henry | 2014-07-26 |   115 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      8 | Henry | 2014-08-01 |   197 |
|      9 | Ben   | 2014-08-02 |    79 |
```

```
|     10 | Henry | 2014-07-30 |   203 |
+--------+-------+------------+-------+
```

But what if you want to display a list and sort it on the basis of a summary value not present in the rows? That's a little trickier. Suppose that you want to show each driver's rows by date, but place those drivers who drive the most miles first. You can't do this with a summary query because then you wouldn't get back the individual driver rows. But you can't do it without a summary query, either, because the summary values are required for sorting. The way out of the dilemma is to create another table containing the summary value per driver and join it to the original table. That way you can produce the individual rows and also sort them by the summary values.

To summarize the driver totals into another table, do this:

```
mysql> CREATE TABLE tmp
    -> SELECT name, SUM(miles) AS driver_miles FROM driver_log GROUP BY name;
```

That produces the values we need to put the names in the proper total-miles order:

```
mysql> SELECT * FROM tmp ORDER BY driver_miles DESC;
+-------+--------------+
| name  | driver_miles |
+-------+--------------+
| Henry |          911 |
| Suzi  |          893 |
| Ben   |          362 |
+-------+--------------+
```

Then use the `name` values to join the summary table to the `driver_log` table, and use the `driver_miles` values to sort the result:

```
mysql> SELECT tmp.driver_miles, driver_log.*
    -> FROM driver_log INNER JOIN tmp ON driver_log.name = tmp.name
    -> ORDER BY tmp.driver_miles DESC, driver_log.trav_date;
+--------------+--------+-------+------------+-------+
| driver_miles | rec_id | name  | trav_date  | miles |
+--------------+--------+-------+------------+-------+
|          911 |      6 | Henry | 2014-07-26 |   115 |
|          911 |      4 | Henry | 2014-07-27 |    96 |
|          911 |      3 | Henry | 2014-07-29 |   300 |
|          911 |     10 | Henry | 2014-07-30 |   203 |
|          911 |      8 | Henry | 2014-08-01 |   197 |
|          893 |      2 | Suzi  | 2014-07-29 |   391 |
|          893 |      7 | Suzi  | 2014-08-02 |   502 |
|          362 |      5 | Ben   | 2014-07-29 |   131 |
|          362 |      1 | Ben   | 2014-07-30 |   152 |
|          362 |      9 | Ben   | 2014-08-02 |    79 |
+--------------+--------+-------+------------+-------+
```

The preceding statement shows the mileage totals in the result. That's only to clarify how the values are being sorted. It's not actually necessary to display them; they're needed only for the `ORDER BY` clause.

To avoid using the temporary table, select the same rows using a subquery in the FROM clause:

```
SELECT tmp.driver_miles, driver_log.*
FROM driver_log INNER JOIN
(SELECT name, SUM(miles) AS driver_miles
FROM driver_log GROUP BY name) AS tmp
ON driver_log.name = tmp.name
ORDER BY tmp.driver_miles DESC, driver_log.trav_date;
```

14.10. Referring to Join Output Column Names in Programs

Problem

You need to process the result of a join from within a program, but column names in the result set aren't unique.

Solution

Rewrite the query using column aliases so that each column has a unique name. Alternatively, refer to the columns by position.

Discussion

Joins typically retrieve columns from related tables and it's not unusual for columns selected from different tables to have the same names. Consider the following join that shows the items in your art collection. For each painting, it displays artist name, painting title, the state in which you acquired the item, and its price:

```
mysql> SELECT artist.name, painting.title, states.name, painting.price
    -> FROM artist INNER JOIN painting INNER JOIN states
    -> ON artist.a_id = painting.a_id AND painting.state = states.abbrev;
+----------+------------------+----------+-------+
| name     | title            | name     | price |
+----------+------------------+----------+-------+
| Da Vinci | The Last Supper  | Indiana  |    34 |
| Da Vinci | Mona Lisa        | Michigan |    87 |
| Van Gogh | Starry Night     | Kentucky |    48 |
| Van Gogh | The Potato Eaters | Kentucky |    67 |
| Renoir   | Les Deux Soeurs  | Nebraska |    64 |
+----------+------------------+----------+-------+
```

The statement uses table qualifiers for each output column, but MySQL doesn't include table names in the column headings, so not all column names in the output are distinct. If you process the join result from within a program and fetch rows into a data structure

that references column values by name, nonunique column names cause values to become inaccessible. Suppose that you fetch rows in a Perl DBI script like this:

```
while (my $ref = $sth->fetchrow_hashref ())
{
  ... process row hash here ...
}
```

Fetching rows into the hash yields three hash elements (`name`, `title`, `price`); one of the `name` elements is lost. To solve this problem, supply aliases that make the column names unique:

```
SELECT artist.name AS painter, painting.title,
  states.name AS state, painting.price
FROM artist INNER JOIN painting INNER JOIN states
ON artist.a_id = painting.a_id AND painting.state = states.abbrev;
```

Now fetching rows into a hash yields four hash elements (`painter`, `title`, `state`, `price`).

To address the problem without column renaming, fetch the row into something other than a hash. For example, fetch the row into an array and refer to the columns by ordinal position within the array:

```
while (my @val = $sth->fetchrow_array ())
{
  print "painter: $val[0], title: $val[1], "
        . "state: $val[2], price: $val[3]\n";
}
```

CHAPTER 15
Statistical Techniques

15.0. Introduction

This chapter covers several topics that relate to basic statistical techniques. For the most part, these recipes build on those described in earlier chapters, such as the summary techniques discussed in Chapter 8, and join techniques from Chapter 14. The examples here thus show additional ways to apply the material from those chapters. Broadly speaking, the topics discussed in this chapter include:

- Techniques for characterizing a dataset, such as calculating descriptive statistics, generating frequency distributions, counting missing values, and calculating least-squares regressions or correlation coefficients
- Randomization methods, such as how to generate random numbers and apply them to randomizing a set of rows or to selecting individual items randomly from the rows
- Techniques for calculating successive-observation differences, cumulative sums, and running averages.
- Methods for producing rank assignments and generating team standings

Statistics covers such a large and diverse array of topics that this chapter necessarily only scratches the surface and simply illustrates a few of the potential areas in which MySQL may be applied to statistical analysis. Note that some statistical measures can be defined in different ways (for example, do you calculate standard deviation based on *n* degrees of freedom, or *n*–1?). If the definition I use for a given term doesn't match the one you prefer, adapt the queries or algorithms shown here appropriately.

You can find scripts related to the examples discussed here in the *stats* directory of the `recipes` distribution, and scripts for creating example tables in the *tables* directory.

15.1. Calculating Descriptive Statistics

Problem

You want to characterize a dataset by computing general descriptive or summary statistics.

Solution

Many common descriptive statistics, such as mean and standard deviation, are obtained by applying aggregate functions to your data. Others, such as median or mode, are calculated based on counting queries.

Discussion

Suppose that a `testscore` table contains observations representing subject ID, age, sex, and test score:

```
mysql> SELECT subject, age, sex, score FROM testscore ORDER BY subject;
+---------+-----+-----+-------+
| subject | age | sex | score |
+---------+-----+-----+-------+
|       1 |   5 | M   |     5 |
|       2 |   5 | M   |     4 |
|       3 |   5 | F   |     6 |
|       4 |   5 | F   |     7 |
|       5 |   6 | M   |     8 |
|       6 |   6 | M   |     9 |
|       7 |   6 | F   |     4 |
|       8 |   6 | F   |     6 |
|       9 |   7 | M   |     8 |
|      10 |   7 | M   |     6 |
|      11 |   7 | F   |     9 |
|      12 |   7 | F   |     7 |
|      13 |   8 | M   |     9 |
|      14 |   8 | M   |     6 |
|      15 |   8 | F   |     7 |
|      16 |   8 | F   |    10 |
|      17 |   9 | M   |     9 |
|      18 |   9 | M   |     7 |
|      19 |   9 | F   |    10 |
|      20 |   9 | F   |     9 |
+---------+-----+-----+-------+
```

A good first step in analyzing a set of observations is to generate some descriptive statistics that summarize their general characteristics as a whole. Common statistical values of this kind include:

- The number of observations, their sum, and their range (minimum and maximum)

- Measures of central tendency, such as mean, median, and mode
- Measures of variation, such as standard deviation and variance

Aside from the median and mode, all of these can be calculated easily by invoking aggregate functions:

```
mysql> SELECT COUNT(score) AS n,
    -> SUM(score) AS sum,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> AVG(score) AS mean,
    -> STDDEV_SAMP(score) AS 'std. dev.',
    -> VAR_SAMP(score) AS 'variance'
    -> FROM testscore;
+----+------+---------+---------+--------+-----------+----------+
| n  | sum  | minimum | maximum | mean   | std. dev. | variance |
+----+------+---------+---------+--------+-----------+----------+
| 20 |  146 |       4 |      10 | 7.3000 |    1.8382 |   3.3789 |
+----+------+---------+---------+--------+-----------+----------+
```

The `STDDEV_SAMP()` and `VAR_SAMP()` functions produce sample measures rather than population measures. That is, for a set of *n* values, they produce a result that is based on *n*–1 degrees of freedom. For the population measures, which are based on *n* degrees of freedom, use `STDDEV_POP()` and `VAR_POP()` instead. `STDDEV()` and `VARIANCE()` are synonyms for `STDDEV_POP()` and `VAR_POP()`.

Standard deviation can be used to identify outliers—values that are uncharacteristically far from the mean. For example, to select values that lie more than three standard deviations from the mean, do this:

```
SELECT @mean := AVG(score), @std := STDDEV_SAMP(score) FROM testscore;
SELECT score FROM testscore WHERE ABS(score-@mean) > @std * 3;
```

MySQL has no built-in function for computing the mode or median of a set of values, but you can compute them yourself. To determine the mode (the value that occurs most frequently), count each value and see which is most common:

```
mysql> SELECT score, COUNT(score) AS frequency
    -> FROM testscore GROUP BY score ORDER BY frequency DESC;
+-------+-----------+
| score | frequency |
+-------+-----------+
|     9 |         5 |
|     6 |         4 |
|     7 |         4 |
|     4 |         2 |
|     8 |         2 |
|    10 |         2 |
|     5 |         1 |
+-------+-----------+
```

In this case, 9 is the modal score value.

The median of a set of ordered values can be calculated like this:[1]

- If the number of values is odd, the median is the middle value.
- If the number of values is even, the median is the average of the two middle values.

Based on that definition, use the following procedure to determine the median of a set of observations stored in the database:

1. Issue a query to count the number of observations. From the count, you can determine whether the median calculation requires one or two values, and what their indexes are within the ordered set of observations.
2. Issue a query that includes an `ORDER BY` clause to sort the observations and a `LIMIT` clause to pull out the middle value or values.
3. If there is a single middle value, it is the median. Otherwise, take the average of the middle values.

Suppose that a table `t` contains a `score` column with 37 values (an odd number). To get the median, select a single value using a statement like this:

```
SELECT score FROM t ORDER BY score LIMIT 18,1
```

If the column contains 38 values (an even number), select two values:

```
SELECT score FROM t ORDER BY score LIMIT 18,2
```

Then take the values returned by the statement and compute the median from their average.

The following Perl function implements a median calculation. It takes a database handle and the names of the database, table, and column that contain the set of observations. Then it generates the statement that retrieves the relevant values and returns their average:

```
sub median
{
my ($dbh, $db_name, $tbl_name, $col_name) = @_;
my ($count, $limit);

  $db_name = $dbh->quote_identifier ($db_name);
  $tbl_name = $dbh->quote_identifier ($tbl_name);
  $col_name = $dbh->quote_identifier ($col_name);

  $count = $dbh->selectrow_array (qq{
```

1. The definition of median given here isn't fully general; it doesn't address what to do if the middle values in the dataset are duplicated.

```
    SELECT COUNT($col_name) FROM $db_name.$tbl_name
  });
  return undef unless $count > 0;
  if ($count % 2 == 1)  # odd number of values; select middle value
  {
    $limit = sprintf ("LIMIT %d,1", ($count-1)/2);
  }
  else                  # even number of values; select middle two values
  {
    $limit = sprintf ("LIMIT %d,2", $count/2 - 1);
  }

  my $sth = $dbh->prepare (qq{
    SELECT $col_name FROM $db_name.$tbl_name ORDER BY $col_name $limit
  });
  $sth->execute ();
  my ($n, $sum) = (0, 0);
  while (my $ref = $sth->fetchrow_arrayref ())
  {
    ++$n;
    $sum += $ref->[0];
  }
  return $sum / $n;
}
```

The preceding technique works for a set of values stored in the database. If you have already fetched an ordered set of values into an array `@val`, compute the median like this instead:

```
if (@val == 0)          # array is empty, median is undefined
{
  $median = undef;
}
elsif (@val % 2 == 1)   # array size is odd, median is middle number
{
  $median = $val[(@val-1)/2];
}
else                    # array size is even; median is average
{                       # of two middle numbers
  $median = ($val[@val/2 - 1] + $val[@val/2]) / 2;
}
```

The code works for arrays that have an initial subscript of 0; for languages that use 1-based array indexes, adjust the algorithm accordingly.

15.2. Per-Group Descriptive Statistics

Problem

You want to produce descriptive statistics for each subgroup of a set of observations.

Solution

Use aggregate functions, but employ a `GROUP BY` clause to arrange observations into the appropriate groups.

Discussion

Recipe 15.1 shows how to compute descriptive statistics for the entire set of scores in the `testscore` table. To be more specific, use `GROUP BY` to divide the observations into groups and calculate statistics for each of them. For example, the subjects in the `test score` table are listed by age and sex, so it's possible to calculate similar statistics by age or sex (or both) by application of appropriate `GROUP BY` clauses.

Here's how to calculate by age:

```
mysql> SELECT age, COUNT(score) AS n,
    -> SUM(score) AS sum,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> AVG(score) AS mean,
    -> STDDEV_SAMP(score) AS 'std. dev.',
    -> VAR_SAMP(score) AS 'variance'
    -> FROM testscore
    -> GROUP BY age;
+-----+---+------+---------+---------+--------+-----------+----------+
| age | n | sum  | minimum | maximum | mean   | std. dev. | variance |
+-----+---+------+---------+---------+--------+-----------+----------+
|   5 | 4 |   22 |       4 |       7 | 5.5000 |    1.2910 |   1.6667 |
|   6 | 4 |   27 |       4 |       9 | 6.7500 |    2.2174 |   4.9167 |
|   7 | 4 |   30 |       6 |       9 | 7.5000 |    1.2910 |   1.6667 |
|   8 | 4 |   32 |       6 |      10 | 8.0000 |    1.8257 |   3.3333 |
|   9 | 4 |   35 |       7 |      10 | 8.7500 |    1.2583 |   1.5833 |
+-----+---+------+---------+---------+--------+-----------+----------+
```

By sex:

```
mysql> SELECT sex, COUNT(score) AS n,
    -> SUM(score) AS sum,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> AVG(score) AS mean,
    -> STDDEV_SAMP(score) AS 'std. dev.',
    -> VAR_SAMP(score) AS 'variance'
    -> FROM testscore
    -> GROUP BY sex;
+-----+----+------+---------+---------+--------+-----------+----------+
| sex | n  | sum  | minimum | maximum | mean   | std. dev. | variance |
+-----+----+------+---------+---------+--------+-----------+----------+
| M   | 10 |   71 |       4 |       9 | 7.1000 |    1.7920 |   3.2111 |
| F   | 10 |   75 |       4 |      10 | 7.5000 |    1.9579 |   3.8333 |
+-----+----+------+---------+---------+--------+-----------+----------+
```

By age and sex:

```
mysql> SELECT age, sex, COUNT(score) AS n,
    -> SUM(score) AS sum,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> AVG(score) AS mean,
    -> STDDEV_SAMP(score) AS 'std. dev.',
    -> VAR_SAMP(score) AS 'variance'
    -> FROM testscore
    -> GROUP BY age, sex;
+-----+-----+---+------+---------+---------+--------+-----------+----------+
| age | sex | n | sum  | minimum | maximum | mean   | std. dev. | variance |
+-----+-----+---+------+---------+---------+--------+-----------+----------+
|   5 | M   | 2 |    9 |       4 |       5 | 4.5000 |    0.7071 |   0.5000 |
|   5 | F   | 2 |   13 |       6 |       7 | 6.5000 |    0.7071 |   0.5000 |
|   6 | M   | 2 |   17 |       8 |       9 | 8.5000 |    0.7071 |   0.5000 |
|   6 | F   | 2 |   10 |       4 |       6 | 5.0000 |    1.4142 |   2.0000 |
|   7 | M   | 2 |   14 |       6 |       8 | 7.0000 |    1.4142 |   2.0000 |
|   7 | F   | 2 |   16 |       7 |       9 | 8.0000 |    1.4142 |   2.0000 |
|   8 | M   | 2 |   15 |       6 |       9 | 7.5000 |    2.1213 |   4.5000 |
|   8 | F   | 2 |   17 |       7 |      10 | 8.5000 |    2.1213 |   4.5000 |
|   9 | M   | 2 |   16 |       7 |       9 | 8.0000 |    1.4142 |   2.0000 |
|   9 | F   | 2 |   19 |       9 |      10 | 9.5000 |    0.7071 |   0.5000 |
+-----+-----+---+------+---------+---------+--------+-----------+----------+
```

15.3. Generating Frequency Distributions

Problem

You want to know the frequency of occurrence for each value in a table.

Solution

Derive a frequency distribution that summarizes the contents of your dataset.

Discussion

A common application for per-group summary techniques is to generate a *frequency distribution* that shows how often each value occurs. For the `testscore` table, the frequency distribution looks like this:

```
mysql> SELECT score, COUNT(score) AS counts
    -> FROM testscore GROUP BY score;
+-------+--------+
| score | counts |
+-------+--------+
|     4 |      2 |
|     5 |      1 |
|     6 |      4 |
```

```
|     7 |      4 |
|     8 |      2 |
|     9 |      5 |
|    10 |      2 |
+-------+--------+
```

Expressing the results in percentages rather than counts yields relative frequency distribution. To show each count as a percentage of the total, use one query to get the total number of observations and another to calculate the percentages for each group:

```
mysql> SET @n = (SELECT COUNT(score) FROM testscore);
mysql> SELECT score, (COUNT(score)*100)/@n AS percent
    -> FROM testscore GROUP BY score;
+-------+---------+
| score | percent |
+-------+---------+
|     4 | 10.0000 |
|     5 |  5.0000 |
|     6 | 20.0000 |
|     7 | 20.0000 |
|     8 | 10.0000 |
|     9 | 25.0000 |
|    10 | 10.0000 |
+-------+---------+
```

The distributions just shown summarize the number of values for individual scores. However, if the dataset contains a large number of distinct values and you want a distribution that shows only a small number of categories, you may want to lump values into categories and produce a count for each category. Recipe 8.10 discusses "lumping" techniques.

One typical use of frequency distributions is to export the results for use in a graphing program. But MySQL itself can generate a simple ASCII chart as a visual representation of the distribution. To display an ASCII bar chart of the test score counts, convert the counts to strings of * characters:

```
mysql> SELECT score, REPEAT('*',COUNT(score)) AS 'count histogram'
    -> FROM testscore GROUP BY score;
+-------+-----------------+
| score | count histogram |
+-------+-----------------+
|     4 | **              |
|     5 | *               |
|     6 | ****            |
|     7 | ****            |
|     8 | **              |
|     9 | *****           |
|    10 | **              |
+-------+-----------------+
```

To chart the relative frequency distribution instead, use the percentage values:

```
mysql> SET @n = (SELECT COUNT(score) FROM testscore);
mysql> SELECT score,
    -> REPEAT('*',(COUNT(score)*100)/@n) AS 'percent histogram'
    -> FROM testscore GROUP BY score;
+-------+--------------------------+
| score | percent histogram        |
+-------+--------------------------+
|     4 | **********               |
|     5 | *****                    |
|     6 | ********************     |
|     7 | ********************     |
|     8 | **********               |
|     9 | ************************ |
|    10 | **********               |
+-------+--------------------------+
```

The ASCII chart method is crude, obviously, but it's a quick way to get a picture of the distribution of observations and requires no other tools.

If you generate a frequency distribution for a range of categories where some of the categories are not represented in your observations, the missing categories do not appear in the output. To force each category to be displayed, use a reference table and a `LEFT JOIN` (a technique discussed in Recipe 14.8). For the `testscore` table, the possible scores range from 0 to 10, so a reference table should contain each of those values:

```
mysql> CREATE TABLE ref (score INT);
mysql> INSERT INTO ref (score)
    -> VALUES(0),(1),(2),(3),(4),(5),(6),(7),(8),(9),(10);
```

Then join the reference table to the test scores to generate the frequency distribution. This query shows the counts as well as the histogram:

```
mysql> SELECT ref.score, COUNT(testscore.score) AS counts,
    -> REPEAT('*',COUNT(testscore.score)) AS 'count histogram'
    -> FROM ref LEFT JOIN testscore ON ref.score = testscore.score
    -> GROUP BY ref.score;
+-------+--------+-----------+
| score | counts | histogram |
+-------+--------+-----------+
|     0 |      0 |           |
|     1 |      0 |           |
|     2 |      0 |           |
|     3 |      0 |           |
|     4 |      2 | **        |
|     5 |      1 | *         |
|     6 |      4 | ****      |
|     7 |      4 | ****      |
|     8 |      2 | **        |
|     9 |      5 | *****     |
|    10 |      2 | **        |
+-------+--------+-----------+
```

This distribution includes rows for scores 0 through 3, none of which appear in the frequency distribution shown earlier.

The same principle applies to relative frequency distributions:

```
mysql> SET @n = (SELECT COUNT(score) FROM testscore);
mysql> SELECT ref.score, (COUNT(testscore.score)*100)/@n AS percent,
    -> REPEAT('*',(COUNT(testscore.score)*100)/@n) AS 'percent histogram'
    -> FROM ref LEFT JOIN testscore ON ref.score = testscore.score
    -> GROUP BY ref.score;
+-------+---------+--------------------------+
| score | percent | percent histogram        |
+-------+---------+--------------------------+
|     0 |  0.0000 |                          |
|     1 |  0.0000 |                          |
|     2 |  0.0000 |                          |
|     3 |  0.0000 |                          |
|     4 | 10.0000 | **********               |
|     5 |  5.0000 | *****                    |
|     6 | 20.0000 | ********************     |
|     7 | 20.0000 | ********************     |
|     8 | 10.0000 | **********               |
|     9 | 25.0000 | ************************ |
|    10 | 10.0000 | **********               |
+-------+---------+--------------------------+
```

15.4. Counting Missing Values

Problem

A set of observations is incomplete. You want to find out how much so.

Solution

Count the number of `NULL` values in the set.

Discussion

Values can be missing from a set of observations for any number of reasons: a test may not yet have been administered, something may have gone wrong during the test that requires invalidating the observation, and so forth. You can represent such observations in a dataset as `NULL` values to signify that they're missing or otherwise invalid, then use summary statements to characterize the completeness of the dataset.

If a table `t` contains values to be summarized along a single dimension, a simple summary suffices to characterize the missing values. Suppose that `t` looks like this:

```
mysql> SELECT subject, score FROM t ORDER BY subject;
+---------+-------+
| subject | score |
+---------+-------+
|       1 |    38 |
|       2 |  NULL |
|       3 |    47 |
|       4 |  NULL |
|       5 |    37 |
|       6 |    45 |
|       7 |    54 |
|       8 |  NULL |
|       9 |    40 |
|      10 |    49 |
+---------+-------+
```

`COUNT(*)` counts the total number of rows, and `COUNT(score)` counts the number of nonmissing scores. The difference between the two values is the number of missing scores, and that difference in relation to the total provides the percentage of missing scores. Perform these calculations as follows:

```
mysql> SELECT COUNT(*) AS 'n (total)',
    -> COUNT(score) AS 'n (nonmissing)',
    -> COUNT(*) - COUNT(score) AS 'n (missing)',
    -> ((COUNT(*) - COUNT(score)) * 100) / COUNT(*) AS '% missing'
    -> FROM t;
+-----------+---------------+-------------+-----------+
| n (total) | n (nonmissing) | n (missing) | % missing |
+-----------+---------------+-------------+-----------+
|        10 |              7 |           3 |   30.0000 |
+-----------+---------------+-------------+-----------+
```

As an alternative to counting `NULL` values as the difference between counts, count them directly using `SUM(ISNULL(score))`. The `ISNULL()` function returns 1 if its argument is `NULL`, zero otherwise:

```
mysql> SELECT COUNT(*) AS 'n (total)',
    -> COUNT(score) AS 'n (nonmissing)',
    -> SUM(ISNULL(score)) AS 'n (missing)',
    -> (SUM(ISNULL(score)) * 100) / COUNT(*) AS '% missing'
    -> FROM t;
+-----------+---------------+-------------+-----------+
| n (total) | n (nonmissing) | n (missing) | % missing |
+-----------+---------------+-------------+-----------+
|        10 |              7 |           3 |   30.0000 |
+-----------+---------------+-------------+-----------+
```

If values are arranged in groups, occurrences of `NULL` values can be assessed on a per-group basis. Suppose that `t` contains scores for subjects that are distributed among conditions for two factors A and B, each of which has two levels:

```
mysql> SELECT subject, A, B, score FROM t ORDER BY subject;
+---------+------+------+-------+
| subject | A    | B    | score |
+---------+------+------+-------+
|       1 |    1 |    1 |    18 |
|       2 |    1 |    1 |  NULL |
|       3 |    1 |    1 |    23 |
|       4 |    1 |    1 |    24 |
|       5 |    1 |    2 |    17 |
|       6 |    1 |    2 |    23 |
|       7 |    1 |    2 |    29 |
|       8 |    1 |    2 |    32 |
|       9 |    2 |    1 |    17 |
|      10 |    2 |    1 |  NULL |
|      11 |    2 |    1 |  NULL |
|      12 |    2 |    1 |    25 |
|      13 |    2 |    2 |  NULL |
|      14 |    2 |    2 |    33 |
|      15 |    2 |    2 |    34 |
|      16 |    2 |    2 |    37 |
+---------+------+------+-------+
```

To produce a summary for each combination of conditions, use a `GROUP BY` clause:

```
mysql> SELECT A, B, COUNT(*) AS 'n (total)',
    -> COUNT(score) AS 'n (nonmissing)',
    -> COUNT(*) - COUNT(score) AS 'n (missing)',
    -> ((COUNT(*) - COUNT(score)) * 100) / COUNT(*) AS '% missing'
    -> FROM t
    -> GROUP BY A, B;
+------+------+-----------+---------------+-------------+-----------+
| A    | B    | n (total) | n (nonmissing) | n (missing) | % missing |
+------+------+-----------+---------------+-------------+-----------+
|    1 |    1 |         4 |             3 |           1 |   25.0000 |
|    1 |    2 |         4 |             4 |           0 |    0.0000 |
|    2 |    1 |         4 |             2 |           2 |   50.0000 |
|    2 |    2 |         4 |             3 |           1 |   25.0000 |
+------+------+-----------+---------------+-------------+-----------+
```

15.5. Calculating Linear Regressions or Correlation Coefficients

Problem

You want to calculate the least-squares regression line for two variables or the correlation coefficient that expresses the strength of the relationship between them.

Solution

Apply summary functions to calculate the necessary terms.

Discussion

When the data values for two variables X and Y are stored in a database, the least-squares regression for them can be calculated easily using aggregate functions. The same is true for the correlation coefficient. The two calculations are actually fairly similar, and many terms for performing the computations are common to the two procedures.

Suppose that you want to calculate a least-squares regression using the age and test score values for the observations in the `testscore` table:

```
mysql> SELECT age, score FROM testscore;
+-----+-------+
| age | score |
+-----+-------+
|   5 |     5 |
|   5 |     4 |
|   5 |     6 |
|   5 |     7 |
|   6 |     8 |
|   6 |     9 |
|   6 |     4 |
|   6 |     6 |
|   7 |     8 |
|   7 |     6 |
|   7 |     9 |
|   7 |     7 |
|   8 |     9 |
|   8 |     6 |
|   8 |     7 |
|   8 |    10 |
|   9 |     9 |
|   9 |     7 |
|   9 |    10 |
|   9 |     9 |
+-----+-------+
```

The following equation expresses the regression line, where `a` and `b` are the intercept and slope of the line:

$$Y = bX + a$$

Letting `age` be *X* and `score` be *Y*, begin by computing the terms needed for the regression equation. These include the number of observations; the means, sums, and sums of squares for each variable; and the sum of the products of each variable:[2]

```
mysql> SELECT
    -> @n := COUNT(score) AS N,
    -> @meanX := AVG(age) AS 'X mean',
    -> @sumX := SUM(age) AS 'X sum',
```

2. To see where these terms come from, consult any standard statistics text.

```
    -> @sumXX := SUM(age*age) AS 'X sum of squares',
    -> @meanY := AVG(score) AS 'Y mean',
    -> @sumY := SUM(score) AS 'Y sum',
    -> @sumYY := SUM(score*score) AS 'Y sum of squares',
    -> @sumXY := SUM(age*score) AS 'X*Y sum'
    -> FROM testscore\G
*************************** 1. row ***************************
              N: 20
         X mean: 7.000000000
          X sum: 140
X sum of squares: 1020
         Y mean: 7.300000000
          Y sum: 146
Y sum of squares: 1130
        X*Y sum: 1053
```

From those terms, calculate the regression slope and intercept as follows:

```
mysql> SET @b := (@n*@sumXY - @sumX*@sumY) / (@n*@sumXX - @sumX*@sumX);
mysql> SET @a := (@meanY - @b*@meanX);
mysql> SELECT @b AS slope, @a AS intercept;
+-------------+---------------------+
| slope       | intercept           |
+-------------+---------------------+
| 0.775000000 | 1.875000000000000000 |
+-------------+---------------------+
```

The regression equation then is:

```
mysql> SELECT CONCAT('Y = ',@b,'X + ',@a) AS 'least-squares regression';
+---------------------------------------+
| least-squares regression              |
+---------------------------------------+
| Y = 0.775000000X + 1.875000000000000000 |
+---------------------------------------+
```

To compute the correlation coefficient, use many of the same terms:

```
mysql> SELECT
    -> (@n*@sumXY - @sumX*@sumY)
    -> / SQRT((@n*@sumXX - @sumX*@sumX) * (@n*@sumYY - @sumY*@sumY))
    -> AS correlation;
+--------------------+
| correlation        |
+--------------------+
| 0.6117362044219903 |
+--------------------+
```

15.6. Generating Random Numbers

Problem

You need a source of random numbers.

Solution

Use the `RAND()` function.

Discussion

MySQL has a `RAND()` function that produces random numbers between 0 and 1:

```
mysql> SELECT RAND(), RAND(), RAND();
+--------------------+--------------------+--------------------+
| RAND()             | RAND()             | RAND()             |
+--------------------+--------------------+--------------------+
| 0.37415416573561183 | 0.9068914557871329 | 0.41199481246247405 |
+--------------------+--------------------+--------------------+
```

When invoked with an integer argument, `RAND()` uses that value to seed the random number generator. You can use this feature to produce a repeatable series of numbers for a column of a query result. The following example shows that `RAND()` without an argument produces a different column of values per query, whereas `RAND(`*`N`*`)` produces a repeatable column:

```
mysql> SELECT i, RAND(), RAND(10), RAND(20) FROM t;
+------+--------------------+--------------------+--------------------+
| i    | RAND()             | RAND(10)           | RAND(20)           |
+------+--------------------+--------------------+--------------------+
|    1 | 0.00708185882035816 |  0.6570515219653505 | 0.15888261251047497 |
|    2 |  0.5417692908474889 | 0.12820613023657923 |  0.6355305003333189 |
|    3 |  0.6876009085100152 |  0.6698761160204896 |  0.7010046948688149 |
|    4 |  0.8126967007412544 |  0.9647622201263553 |  0.5984320040777623 |
+------+--------------------+--------------------+--------------------+
mysql> SELECT i, RAND(), RAND(10), RAND(20) FROM t;
+------+---------------------+--------------------+--------------------+
| i    | RAND()              | RAND(10)           | RAND(20)           |
+------+---------------------+--------------------+--------------------+
|    1 | 0.059957268703689115 |  0.6570515219653505 | 0.15888261251047497 |
|    2 |   0.9068000166740269 | 0.12820613023657923 |  0.6355305003333189 |
|    3 |  0.35412830799271194 |  0.6698761160204896 |  0.7010046948688149 |
|    4 | 0.050241520675124156 |  0.9647622201263553 |  0.5984320040777623 |
+------+---------------------+--------------------+--------------------+
```

To seed `RAND()` randomly, pick a seed value based on a source of entropy. Possible sources are the current timestamp or connection identifier, alone or perhaps in combination:

```
RAND(UNIX_TIMESTAMP())
RAND(CONNECTION_ID())
RAND(UNIX_TIMESTAMP()+CONNECTION_ID())
```

However, it's probably better to use other seed value sources if you have them. For example, if your system has a */dev/random* or */dev/urandom* device, read the device and use it to generate a value for seeding `RAND()`.

How Random Is RAND()?

Does the `RAND()` function generate evenly distributed numbers? Check it out for yourself with the following Python script, *rand_test.py*, from the *stats* directory of the `recipes` distribution. (That directory also contains equivalent scripts in other languages.) The script uses `RAND()` to generate random numbers and constructs a frequency distribution from them, using 10 categories ("buckets"). This provides a means of assessing how evenly distributed the values are:

```
#!/usr/bin/python
# rand_test.pl: create a frequency distribution of RAND() values.
# This provides a test of the randomness of RAND().

# Method: Draw random numbers in the range from 0 to 1.0,
# and count how many of them occur in .1-sized intervals

import cookbook

npicks = 1000     # number of times to pick a number
bucket = [0] * 10 # buckets for counting picks in each interval

conn = cookbook.connect()
cursor = conn.cursor()

for i in range(0, npicks):
  cursor.execute("SELECT RAND()")
  (val,) = cursor.fetchone()
  slot = int(val * 10)
  if slot > 9:
    slot = 9    # put 1.0 in last slot
  bucket[slot] += 1

cursor.close()
conn.close()

# Print the resulting frequency distribution

for slot, val in enumerate(bucket):
  print("%2d  %d" % (slot+1, val))
```

15.7. Randomizing a Set of Rows

Problem

You want to randomize a set of rows or values.

Solution

Use `ORDER BY RAND()`.

Discussion

MySQL's `RAND()` function can be used to randomize the order in which a query returns its rows. Somewhat paradoxically, this randomization is achieved by adding an `ORDER BY` clause to the query. The technique is roughly equivalent to a spreadsheet randomization method. Suppose that a spreadsheet contains this set of values:

```
Patrick
Penelope
Pertinax
Polly
```

To place these in random order, first add another column that contains randomly chosen numbers:

```
Patrick           .73
Penelope          .37
Pertinax          .16
Polly             .48
```

Then sort the rows according to the values of the random numbers:

```
Pertinax          .16
Penelope          .37
Polly             .48
Patrick           .73
```

At this point, the original values have been placed in random order; the effect of sorting the random numbers is to randomize the values associated with them. To rerandomize the values, choose another set of random numbers, and sort the rows again.

In MySQL, achieve a similar effect by associating a set of random numbers with a query result and sorting the result by those numbers. To do this, add an `ORDER BY RAND()` clause:

```
mysql> SELECT name FROM t ORDER BY RAND();
+----------+
| name     |
+----------+
| Pertinax |
| Patrick  |
| Polly    |
```

```
| Penelope |
+----------+
mysql> SELECT name FROM t ORDER BY RAND();
+----------+
| name     |
+----------+
| Polly    |
| Pertinax |
| Penelope |
| Patrick  |
+----------+
```

Applications for randomizing a set of rows include any scenario that uses selection without replacement (choosing each item from a set of items until there are no more items left). Some examples of this are:

- Determining the starting order for participants in an event. List the participants in a table, and select them in random order.
- Assigning starting lanes or gates to participants in a race. List the lanes in a table, and select a random lane order.
- Choosing the order in which to present a set of quiz questions.
- Shuffling a deck of cards. Represent each card by a row in a table, and shuffle the deck by selecting the rows in random order. Deal them one by one until the deck is exhausted.

To use the last example as an illustration, let's implement a card deck-shuffling algorithm. Shuffling and dealing cards is randomization plus selection without replacement: each card is dealt once before any is dealt twice; when the deck is used up, it is reshuffled to rerandomize it for a new dealing order. Within a program, this task can be performed with MySQL using a table named `deck` that has 52 rows, assuming a set of cards with each combination of 13 face values and 4 suits:

1. Select the entire table, and store it into an array.
2. Each time a card is needed, take the next element from the array.
3. When the array is exhausted, all the cards have been dealt. "Reshuffle" the table to generate a new card order.

Setting up the `deck` table is a tedious task if you insert the 52 card records by writing all the `INSERT` statements manually. The `deck` contents can be generated more easily in combinatorial fashion within a program by generating each pairing of face value with suit. Here's some PHP code that creates a `deck` table with `face` and `suit` columns, then populates the table using nested loops to generate the pairings for the `INSERT` statements:

```
$sth = $dbh->exec ("DROP TABLE IF EXISTS deck");
```

```
$sth = $dbh->exec ("
  CREATE TABLE deck
  (
    face  ENUM('A', 'K', 'Q', 'J', '10', '9', '8',
               '7', '6', '5', '4', '3', '2') NOT NULL,
    suit  ENUM('hearts', 'diamonds', 'clubs', 'spades') NOT NULL
  )
");

$face_array = array ("A", "K", "Q", "J", "10", "9", "8",
                     "7", "6", "5", "4", "3", "2");
$suit_array = array ("hearts", "diamonds", "clubs", "spades");

# insert a "card" into the deck for each combination of suit and face

$sth = $dbh->prepare ("INSERT INTO deck (face,suit) VALUES(?,?)");
foreach ($face_array as $face)
  foreach ($suit_array as $suit)
    $sth->execute (array ($face, $suit));
```

Shuffling the cards is a matter of issuing this statement:

```
SELECT face, suit FROM deck ORDER BY RAND();
```

To do that and store the results in an array within a script, write a `shuffle_deck()` function that issues the query and returns the resulting values in an array (again shown in PHP):

```
function shuffle_deck ($dbh)
{
  $sth = $dbh->query ("SELECT face, suit FROM deck ORDER BY RAND()");
  $sth->setFetchMode (PDO::FETCH_OBJ);
  return ($sth->fetchAll ());
}
```

Deal the cards by keeping a counter that ranges from 0 to 51 to indicate which card to select. When the counter reaches 52, the deck is exhausted and should be shuffled again.

15.8. Selecting Random Items from a Set of Rows

Problem

You want to pick an item or items randomly from a set of values.

Solution

Randomize the values, then pick the first one (or the first few, if you need more than one).

Discussion

If a set of items is stored in MySQL, choose one at random as follows:

1. Select the items in the set in random order, using `ORDER BY RAND()` as described in Recipe 15.7.
2. Add `LIMIT 1` to the query to pick the first item.

For example, to perform a simple simulation of tossing a die, create a `die` table containing rows with values from 1 to 6 corresponding to the six faces of a die cube:

```
CREATE TABLE die (n INT);
```

Then pick rows from the table at random:

```
mysql> SELECT n FROM die ORDER BY RAND() LIMIT 1;
+------+
| n    |
+------+
|    6 |
+------+
mysql> SELECT n FROM die ORDER BY RAND() LIMIT 1;
+------+
| n    |
+------+
|    4 |
+------+
mysql> SELECT n FROM die ORDER BY RAND() LIMIT 1;
+------+
| n    |
+------+
|    5 |
+------+
mysql> SELECT n FROM die ORDER BY RAND() LIMIT 1;
+------+
| n    |
+------+
|    4 |
+------+
```

As you repeat this operation, you pick a random sequence of items from the set. This is a form of selection with replacement: an item is chosen from a pool of items and then returned to the pool for the next pick. Because items are replaced, it's possible to pick the same item multiple times when making successive choices this way. Other examples of selection with replacement include:

- Selecting a banner ad to display on a web page
- Picking a row for a "quote of the day" application

- "Pick a card, any card" magic tricks that begin with a full deck of cards each time

To pick more than one item, change the `LIMIT` argument. For example, to draw five winning entries at random from a table named `drawing` that contains contest entries, use `RAND()` in combination with `LIMIT`:

```
SELECT * FROM drawing ORDER BY RAND() LIMIT 5;
```

A special case occurs when you pick a single row from a table that you know contains a column with values in the range from 1 to *n* in unbroken sequence. Under these circumstances, it's possible to avoid performing an `ORDER BY` operation on the entire table. Pick a random number in that range and select the matching row:

```
SET @id = FLOOR(RAND()*n)+1;
SELECT ... FROM tbl_name WHERE id = @id;
```

This is much quicker than `ORDER BY RAND() LIMIT 1` as the table size increases.

15.9. Calculating Successive-Row Differences

Problem

A table contains successive cumulative values in its rows, and you want to compute the differences between pairs of successive rows.

Solution

Use a self-join that matches pairs of adjacent rows and calculates the differences between members of each pair.

Discussion

Self-joins are useful when you have a set of absolute (or cumulative) values that you want to convert to relative values representing the differences between successive pairs of rows. For example, if you take an automobile trip and write down the total miles traveled at each stopping point, you can compute the difference between successive points to determine the distance from one stop to the next. Here is such a table that shows the stops for a trip from San Antonio, Texas to Madison, Wisconsin. Each row shows the total miles driven as of each stop:

```
mysql> SELECT seq, city, miles FROM trip_log ORDER BY seq;
+-----+------------------+-------+
| seq | city             | miles |
+-----+------------------+-------+
|   1 | San Antonio, TX  |     0 |
|   2 | Dallas, TX       |   263 |
|   3 | Benton, AR       |   566 |
|   4 | Memphis, TN      |   745 |
```

```
|   5 | Portageville, MO |   878 |
|   6 | Champaign, IL    |  1164 |
|   7 | Madison, WI      |  1412 |
+-----+------------------+-------+
```

A self-join can convert these cumulative values to successive differences that represent the distances from each city to the next. The following statement shows how to use the sequence numbers in the rows to match pairs of successive rows and compute the differences between each pair of mileage values:

```
mysql> SELECT t1.seq AS seq1, t2.seq AS seq2,
    -> t1.city AS city1, t2.city AS city2,
    -> t1.miles AS miles1, t2.miles AS miles2,
    -> t2.miles-t1.miles AS dist
    -> FROM trip_log AS t1 INNER JOIN trip_log AS t2
    -> ON t1.seq+1 = t2.seq
    -> ORDER BY t1.seq;
+------+------+------------------+------------------+--------+--------+------+
| seq1 | seq2 | city1            | city2            | miles1 | miles2 | dist |
+------+------+------------------+------------------+--------+--------+------+
|    1 |    2 | San Antonio, TX  | Dallas, TX       |      0 |    263 |  263 |
|    2 |    3 | Dallas, TX       | Benton, AR       |    263 |    566 |  303 |
|    3 |    4 | Benton, AR       | Memphis, TN      |    566 |    745 |  179 |
|    4 |    5 | Memphis, TN      | Portageville, MO |    745 |    878 |  133 |
|    5 |    6 | Portageville, MO | Champaign, IL    |    878 |   1164 |  286 |
|    6 |    7 | Champaign, IL    | Madison, WI      |   1164 |   1412 |  248 |
+------+------+------------------+------------------+--------+--------+------+
```

The presence of the `seq` column in the `trip_log` table is important for calculating successive difference values. It's needed for establishing which row precedes another and matching each row *n* with row *n*+1. The implication is that to perform relative-difference calculations using a table of absolute or cumulative values, it must include a sequence column that has no gaps. If the table contains a sequence column but there are gaps, renumber it (see Recipe 13.5). If the table contains no such column, add one (see Recipe 13.9).

A more complex situation occurs when you compute successive differences for more than one column and use the results in a calculation. The following table, `player_stats`, shows some cumulative numbers for a baseball player at the end of each month of his season. `ab` indicates the total at-bats, and `h` the total hits the player has had as of a given date. (The first row indicates the starting point of the player's season, which is why the `ab` and `h` values are zero.)

```
mysql> SELECT id, date, ab, h, TRUNCATE(IFNULL(h/ab,0),3) AS ba
    -> FROM player_stats ORDER BY id;
+----+------------+-----+----+-------+
| id | date       | ab  | h  | ba    |
+----+------------+-----+----+-------+
|  1 | 2013-04-30 |   0 |  0 | 0.000 |
|  2 | 2013-05-31 |  38 | 13 | 0.342 |
```

```
|  3 | 2013-06-30 | 109 | 31 | 0.284 |
|  4 | 2013-07-31 | 196 | 49 | 0.250 |
|  5 | 2013-08-31 | 304 | 98 | 0.322 |
+----+------------+-----+----+-------+
```

The last column of the query result also shows the player's batting average as of each date. This column is not stored in the table but is easily computed as the ratio of hits to at-bats. The result provides a general idea of how the player's hitting performance changed over the course of the season, but it provides no picture of how the player did during each individual month. To determine that, calculate relative differences between pairs of rows. This is easily done with a self-join that matches row *n* with row *n*+1 to calculate differences between pairs of at-bats and hits values. These differences enable computation of batting average during each month:

```
mysql> SELECT
    -> t1.id AS id1, t2.id AS id2,
    -> t2.date,
    -> t1.ab AS ab1, t2.ab AS ab2,
    -> t1.h AS h1, t2.h AS h2,
    -> t2.ab-t1.ab AS abdiff,
    -> t2.h-t1.h AS hdiff,
    -> TRUNCATE(IFNULL((t2.h-t1.h)/(t2.ab-t1.ab),0),3) AS ba
    -> FROM player_stats AS t1 INNER JOIN player_stats AS t2
    -> ON t1.id+1 = t2.id
    -> ORDER BY t1.id;
+-----+-----+------------+-----+-----+----+----+--------+-------+-------+
| id1 | id2 | date       | ab1 | ab2 | h1 | h2 | abdiff | hdiff | ba    |
+-----+-----+------------+-----+-----+----+----+--------+-------+-------+
|   1 |   2 | 2013-05-31 |   0 |  38 |  0 | 13 |     38 |    13 | 0.342 |
|   2 |   3 | 2013-06-30 |  38 | 109 | 13 | 31 |     71 |    18 | 0.253 |
|   3 |   4 | 2013-07-31 | 109 | 196 | 31 | 49 |     87 |    18 | 0.206 |
|   4 |   5 | 2013-08-31 | 196 | 304 | 49 | 98 |    108 |    49 | 0.453 |
+-----+-----+------------+-----+-----+----+----+--------+-------+-------+
```

These results show much more clearly than the original table that the player started off well but had a slump in the middle of the season, particularly in July. They also indicate just how strong his performance was in August.

15.10. Finding Cumulative Sums and Running Averages

Problem

You have a set of observations measured over time and want to compute the cumulative sum of the observations at each measurement point. Or you want to compute a running average at each point.

Solution

Use a self-join to produce the sets of successive observations at each measurement point, then apply aggregate functions to each set of values to compute its sum or average.

Discussion

Recipe 15.9 illustrates how a self-join can produce relative values from absolute values. A self-join can do the opposite as well, producing cumulative values at each successive stage of a set of observations. The following table shows a set of rainfall measurements taken over a series of days. The values in each row show the observation date and precipitation in inches:

```
mysql> SELECT date, precip FROM rainfall ORDER BY date;
+------------+--------+
| date       | precip |
+------------+--------+
| 2014-06-01 |   1.50 |
| 2014-06-02 |   0.00 |
| 2014-06-03 |   0.50 |
| 2014-06-04 |   0.00 |
| 2014-06-05 |   1.00 |
+------------+--------+
```

To calculate cumulative rainfall for a given day, add that day's precipitation value to the values for all the previous days. For example, determine the cumulative rainfall as of `2014-06-03` like this:

```
mysql> SELECT SUM(precip) FROM rainfall WHERE date <= '2014-06-03';
+-------------+
| SUM(precip) |
+-------------+
|        2.00 |
+-------------+
```

To get the cumulative figures for all days represented in the table, it's tedious to compute the value separately for each day. A self-join can do this for all days with a single statement. Use one instance of the `rainfall` table as a reference, and determine for the date in each row the sum of the `precip` values in all rows occurring up through that date in another instance of the table. The following statement shows the daily and cumulative precipitation for each day:

```
mysql> SELECT t1.date, t1.precip AS 'daily precip',
    -> SUM(t2.precip) AS 'cum. precip'
    -> FROM rainfall AS t1 INNER JOIN rainfall AS t2
    -> ON t1.date >= t2.date
    -> GROUP BY t1.date;
+------------+--------------+-------------+
| date       | daily precip | cum. precip |
+------------+--------------+-------------+
```

```
| 2014-06-01 |         1.50 |        1.50 |
| 2014-06-02 |         0.00 |        1.50 |
| 2014-06-03 |         0.50 |        2.00 |
| 2014-06-04 |         0.00 |        2.00 |
| 2014-06-05 |         1.00 |        3.00 |
+------------+--------------+-------------+
```

The self-join can be extended to display the number of days elapsed at each date, as well as the running averages for amount of precipitation each day:

```
mysql> SELECT t1.date, t1.precip AS 'daily precip',
    -> SUM(t2.precip) AS 'cum. precip',
    -> COUNT(t2.precip) AS 'days elapsed',
    -> AVG(t2.precip) AS 'avg. precip'
    -> FROM rainfall AS t1 INNER JOIN rainfall AS t2
    -> ON t1.date >= t2.date
    -> GROUP BY t1.date;
+------------+--------------+-------------+--------------+-------------+
| date       | daily precip | cum. precip | days elapsed | avg. precip |
+------------+--------------+-------------+--------------+-------------+
| 2014-06-01 |         1.50 |        1.50 |            1 |    1.500000 |
| 2014-06-02 |         0.00 |        1.50 |            2 |    0.750000 |
| 2014-06-03 |         0.50 |        2.00 |            3 |    0.666667 |
| 2014-06-04 |         0.00 |        2.00 |            4 |    0.500000 |
| 2014-06-05 |         1.00 |        3.00 |            5 |    0.600000 |
+------------+--------------+-------------+--------------+-------------+
```

In the preceding statement, the number of days elapsed and the precipitation running averages can be computed easily using `COUNT()` and `AVG()` because there are no missing days in the table. If missing days are permitted, the calculation becomes more complicated because the number of days elapsed for each calculation is no longer the same as the number of rows. You can see this by deleting the rows for the days that had no precipitation to produce "holes" in the table:

```
mysql> DELETE FROM rainfall WHERE precip = 0;
mysql> SELECT date, precip FROM rainfall ORDER BY date;
+------------+--------+
| date       | precip |
+------------+--------+
| 2014-06-01 |   1.50 |
| 2014-06-03 |   0.50 |
| 2014-06-05 |   1.00 |
+------------+--------+
```

Deleting those rows doesn't change the cumulative sum or running average for the dates that remain, but it does change how they must be calculated. If you execute the self-join again, it yields incorrect results for the days-elapsed and average precipitation columns:

```
mysql> SELECT t1.date, t1.precip AS 'daily precip',
    -> SUM(t2.precip) AS 'cum. precip',
    -> COUNT(t2.precip) AS 'days elapsed',
    -> AVG(t2.precip) AS 'avg. precip'
```

```
    -> FROM rainfall AS t1 INNER JOIN rainfall AS t2
    -> ON t1.date >= t2.date
    -> GROUP BY t1.date;
+------------+--------------+-------------+--------------+-------------+
| date       | daily precip | cum. precip | days elapsed | avg. precip |
+------------+--------------+-------------+--------------+-------------+
| 2014-06-01 |         1.50 |        1.50 |            1 |    1.500000 |
| 2014-06-03 |         0.50 |        2.00 |            2 |    1.000000 |
| 2014-06-05 |         1.00 |        3.00 |            3 |    1.000000 |
+------------+--------------+-------------+--------------+-------------+
```

To fix the problem, determine the number of days elapsed a different way. Take the minimum and maximum date involved in each sum and calculate a days-elapsed value from them:

```
DATEDIFF(MAX(t2.date),MIN(t2.date)) + 1
```

That value must be used for the days-elapsed column and for computing the running averages. The resulting statement is as follows:

```
mysql> SELECT t1.date, t1.precip AS 'daily precip',
    -> SUM(t2.precip) AS 'cum. precip',
    -> DATEDIFF(MAX(t2.date),MIN(t2.date)) + 1 AS 'days elapsed',
    -> SUM(t2.precip) / (DATEDIFF(MAX(t2.date),MIN(t2.date)) + 1)
    -> AS 'avg. precip'
    -> FROM rainfall AS t1 INNER JOIN rainfall AS t2
    -> ON t1.date >= t2.date
    -> GROUP BY t1.date;
+------------+--------------+-------------+--------------+-------------+
| date       | daily precip | cum. precip | days elapsed | avg. precip |
+------------+--------------+-------------+--------------+-------------+
| 2014-06-01 |         1.50 |        1.50 |            1 |    1.500000 |
| 2014-06-03 |         0.50 |        2.00 |            3 |    0.666667 |
| 2014-06-05 |         1.00 |        3.00 |            5 |    0.600000 |
+------------+--------------+-------------+--------------+-------------+
```

As this example illustrates, calculation of cumulative values from relative values requires only a column that enables rows to be placed into the proper order. (For the `rainfall` table, that's the `date` column.) Values in the column need not be sequential, or even numeric. This differs from calculations that produce difference values from cumulative values (see Recipe 15.9), which require a table that has a column containing an unbroken sequence.

The running averages in the rainfall examples are based on dividing cumulative precipitation sums by number of days elapsed as of each day. When the table has no gaps, the number of days is the same as the number of values summed, making it easy to find successive averages. When rows are missing, the calculations become more complex. This demonstrates that it's necessary to consider the nature of your data and calculate averages appropriately. The next example is conceptually similar to the previous ones

in that it calculates cumulative sums and running averages, but performs the computations yet another way.

The following table shows a marathon runner's performance at each stage of a 26-kilometer run. The values in each row show the length of each stage in kilometers and how long the runner took to complete the stage. In other words, the values pertain to intervals within the marathon and thus are relative to the whole:

```
mysql> SELECT stage, km, t FROM marathon ORDER BY stage;
+-------+----+----------+
| stage | km | t        |
+-------+----+----------+
|     1 |  5 | 00:15:00 |
|     2 |  7 | 00:19:30 |
|     3 |  9 | 00:29:20 |
|     4 |  5 | 00:17:50 |
+-------+----+----------+
```

To calculate cumulative distance in kilometers at each stage, use a self-join like this:

```
mysql> SELECT t1.stage, t1.km, SUM(t2.km) AS 'cum. km'
    -> FROM marathon AS t1 INNER JOIN marathon AS t2
    -> ON t1.stage >= t2.stage
    -> GROUP BY t1.stage;
+-------+----+---------+
| stage | km | cum. km |
+-------+----+---------+
|     1 |  5 |       5 |
|     2 |  7 |      12 |
|     3 |  9 |      21 |
|     4 |  5 |      26 |
+-------+----+---------+
```

Cumulative distances are easy to compute because they can be summed directly. The calculation for accumulating time values is more involved: convert times to seconds, total the resulting values, and convert the sum back to a time value. To compute the runner's average speed at the end of each stage, take the ratio of cumulative distance over cumulative time. Putting all this together yields the following statement:

```
mysql> SELECT t1.stage, t1.km, t1.t,
    -> SUM(t2.km) AS 'cum. km',
    -> SEC_TO_TIME(SUM(TIME_TO_SEC(t2.t))) AS 'cum. t',
    -> SUM(t2.km)/(SUM(TIME_TO_SEC(t2.t))/(60*60)) AS 'avg. km/hour'
    -> FROM marathon AS t1 INNER JOIN marathon AS t2
    -> ON t1.stage >= t2.stage
    -> GROUP BY t1.stage;
+-------+----+----------+---------+----------+--------------+
| stage | km | t        | cum. km | cum. t   | avg. km/hour |
+-------+----+----------+---------+----------+--------------+
|     1 |  5 | 00:15:00 |       5 | 00:15:00 |      20.0000 |
|     2 |  7 | 00:19:30 |      12 | 00:34:30 |      20.8696 |
|     3 |  9 | 00:29:20 |      21 | 01:03:50 |      19.7389 |
```

```
|     4 |  5 | 00:17:50 |      26 | 01:21:40 |      19.1020 |
+-------+----+----------+---------+----------+--------------+
```

We can see from this that the runner's average pace increased a little during the second stage of the race but then decreased thereafter, presumably as a result of fatigue.

15.11. Assigning Ranks

Problem

You want to assign ranks to a set of values.

Solution

Decide on a ranking method, then put the values in the desired order and apply the method to them.

Discussion

Some kinds of statistical tests require assignment of ranks. This section describes three ranking methods and shows how each can be implemented by using user-defined variables. The examples assume that a table `t` contains the following scores, which are to be ranked with the values in descending order:

```
mysql> SELECT score FROM t ORDER BY score DESC;
+-------+
| score |
+-------+
|     5 |
|     4 |
|     4 |
|     3 |
|     2 |
|     2 |
|     2 |
|     1 |
+-------+
```

Before you use the ideas presented here, be aware that the queries use the *unsanctioned* technique of getting and setting the same user-defined variable within the same statement and therefore might break some day. An alternative (also illustrated here) is to pull the observations into a program that does the ranking calculations.

One type of ranking simply assigns each value its row number within the ordered set of values. To produce such rankings, keep track of the row number and use it for the current rank:

```
mysql> SET @rownum := 0;
mysql> SELECT @rownum := @rownum + 1 AS rank, score
    -> FROM t ORDER BY score DESC;
+------+-------+
| rank | score |
+------+-------+
|    1 |     5 |
|    2 |     4 |
|    3 |     4 |
|    4 |     3 |
|    5 |     2 |
|    6 |     2 |
|    7 |     2 |
|    8 |     1 |
+------+-------+
```

That kind of ranking doesn't take into account the possibility of ties (instances of values that are the same). A second ranking method does so by advancing the rank only when values change:

```
mysql> SET @rank = 0, @prev_val = NULL;
mysql> SELECT @rank := IF(@prev_val=score,@rank,@rank+1) AS rank,
    -> @prev_val := score AS score
    -> FROM t ORDER BY score DESC;
+------+-------+
| rank | score |
+------+-------+
|    1 |     5 |
|    2 |     4 |
|    2 |     4 |
|    3 |     3 |
|    4 |     2 |
|    4 |     2 |
|    4 |     2 |
|    5 |     1 |
+------+-------+
```

A third ranking method is something of a combination of the other two methods. It ranks values by row number, except when ties occur. In that case, the tied values each get a rank equal to the row number of the first of the values. To implement this method, keep track of the row number and the previous value, advancing the rank to the current row number when the value changes:

```
mysql> SET @rownum = 0, @rank = 0, @prev_val = NULL;
mysql> SELECT @rownum := @rownum + 1 AS row,
    -> @rank := IF(@prev_val<>score,@rownum,@rank) AS rank,
    -> @prev_val := score AS score
    -> FROM t ORDER BY score DESC;
+------+------+-------+
| row  | rank | score |
+------+------+-------+
|    1 |    1 |     5 |
```

```
|    2 |    2 |     4 |
|    3 |    2 |     4 |
|    4 |    4 |     3 |
|    5 |    5 |     2 |
|    6 |    5 |     2 |
|    7 |    5 |     2 |
|    8 |    8 |     1 |
+------+------+-------+
```

Ranks are easy to assign within a program as well. For example, the following Ruby fragment ranks the scores in `t` using the third ranking method:

```
dbh.execute("SELECT score FROM t ORDER BY score DESC") do |sth|
  rownum = 0
  rank = 0
  prev_score = nil
  puts "Row\tRank\tScore\n"
  sth.fetch do |row|
    score = row[0]
    rownum += 1
    rank = rownum if rownum == 1 || prev_score != score
    prev_score = score
    puts "#{rownum}\t#{rank}\t#{score}"
  end
end
```

The third type of ranking is commonly used for sporting events. The following table contains the American League pitchers who won 15 or more games during the 2001 baseball season:

```
mysql> SELECT name, wins FROM al_winner ORDER BY wins DESC, name;
+----------------+------+
| name           | wins |
+----------------+------+
| Mulder, Mark   |   21 |
| Clemens, Roger |   20 |
| Moyer, Jamie   |   20 |
| Garcia, Freddy |   18 |
| Hudson, Tim    |   18 |
| Abbott, Paul   |   17 |
| Mays, Joe      |   17 |
| Mussina, Mike  |   17 |
| Sabathia, C.C. |   17 |
| Zito, Barry    |   17 |
| Buehrle, Mark  |   16 |
| Milton, Eric   |   15 |
| Pettitte, Andy |   15 |
| Radke, Brad    |   15 |
| Sele, Aaron    |   15 |
+----------------+------+
```

These pitchers can be assigned ranks using the third method as follows:

```
mysql> SET @rownum = 0, @rank = 0, @prev_val = NULL;
mysql> SELECT @rownum := @rownum + 1 AS row,
    -> @rank := IF(@prev_val<>wins,@rownum,@rank) AS rank,
    -> name,
    -> @prev_val := wins AS wins
    -> FROM al_winner ORDER BY wins DESC;
+------+------+----------------+------+
| row  | rank | name           | wins |
+------+------+----------------+------+
|    1 |    1 | Mulder, Mark   |   21 |
|    2 |    2 | Clemens, Roger |   20 |
|    3 |    2 | Moyer, Jamie   |   20 |
|    4 |    4 | Garcia, Freddy |   18 |
|    5 |    4 | Hudson, Tim    |   18 |
|    6 |    6 | Zito, Barry    |   17 |
|    7 |    6 | Sabathia, C.C. |   17 |
|    8 |    6 | Mussina, Mike  |   17 |
|    9 |    6 | Mays, Joe      |   17 |
|   10 |    6 | Abbott, Paul   |   17 |
|   11 |   11 | Buehrle, Mark  |   16 |
|   12 |   12 | Milton, Eric   |   15 |
|   13 |   12 | Pettitte, Andy |   15 |
|   14 |   12 | Radke, Brad    |   15 |
|   15 |   12 | Sele, Aaron    |   15 |
+------+------+----------------+------+
```

15.12. Computing Team Standings

Problem

You want to compute team standings from their win-loss records, including the games-behind (GB) values.

Solution

Determine which team is in first place, then join that result to the original rows.

Discussion

Standings for sports teams that compete against each other is a ranking problem, but ranks are not based on a single measure as in Recipe 15.11. Standings are based on two values, wins and losses. Teams are ranked according to which has the best win-loss record, and teams not in first place are assigned a "games-behind" value indicating how many games out of first place they are. This section shows how to calculate those values. The first example uses a table containing a single set of team records to illustrate the logic of the calculations. The second example uses a table containing several sets of records (that is, the records for all teams in both divisions of a league, for both halves

of the season). In this case, it's necessary to use a join to perform the calculations independently for each group of teams.

Consider the following table, `standings1`, which contains a single set of baseball team records representing the final standings for the Northern League in the year 1902:

```
mysql> SELECT team, wins, losses FROM standings1
    -> ORDER BY wins-losses DESC;
+-------------+------+--------+
| team        | wins | losses |
+-------------+------+--------+
| Winnipeg    |   37 |     20 |
| Crookston   |   31 |     25 |
| Fargo       |   30 |     26 |
| Grand Forks |   28 |     26 |
| Devils Lake |   19 |     31 |
| Cavalier    |   15 |     32 |
+-------------+------+--------+
```

The rows are sorted by the win-loss differential, which is how to place teams in order from first place to last place. But displays of team standings typically include each team's winning percentage and a figure indicating how many games behind the leader all the other teams are. So let's add that information to the output. Calculating the percentage is easy. It's the ratio of wins to total games played and can be determined using this expression:

```
wins / (wins + losses)
```

This expression involves division by zero when a team has not played any games yet. For simplicity, I'll assume a nonzero number of games. To handle this condition, you'd use a more general expression:

```
IF(wins=0,0,wins/(wins+losses))
```

This expression relies on the fact that no division operation is necessary unless the team has won at least one game.

Determining the games-behind value is a little trickier. It's based on the relationship of the win-loss records for two teams, calculated as the average of two values:

- How many more wins the first-place team has than the second-place team
- How many fewer losses the first-place team has than the second-place team

Suppose that two teams A and B have the following win-loss records:

```
+------+------+--------+
| team | wins | losses |
+------+------+--------+
| A    |   17 |     11 |
| B    |   14 |     12 |
+------+------+--------+
```

Here, team B has to win three more games, and team A has to lose one more game for the teams to be even. The average of three and one is two, thus B is two games behind A. Mathematically, the games-behind calculation for the two teams is:

```
((winsA - winsB) + (lossesB - lossesA)) / 2
```

With a little rearrangement of terms, the expression becomes:

```
((winsA - lossesA) - (winsB - lossesB)) / 2
```

The second expression is equivalent to the first, but it has each factor written as a single team's win-loss differential, rather than as a comparison between teams. That makes it easier to work with because each factor can be determined independently from a single team record. The first factor represents the first-place team's win-loss differential, so if we calculate that value first, the other team GB values can be determined in relation to it.

The first-place team is the one with the largest win-loss differential. To find that value and save it in a variable, use this statement:

```
mysql> SET @wl_diff = (SELECT MAX(wins-losses) FROM standings1);
```

Then use the differential as follows to produce team standings that include winning percentage and GB values:

```
mysql> SELECT team, wins AS W, losses AS L,
    -> wins/(wins+losses) AS PCT,
    -> (@wl_diff - (wins-losses)) / 2 AS GB
    -> FROM standings1
    -> ORDER BY wins-losses DESC, PCT DESC;
+-------------+------+------+--------+---------+
| team        | W    | L    | PCT    | GB      |
+-------------+------+------+--------+---------+
| Winnipeg    |   37 |   20 | 0.6491 |  0.0000 |
| Crookston   |   31 |   25 | 0.5536 |  5.5000 |
| Fargo       |   30 |   26 | 0.5357 |  6.5000 |
| Grand Forks |   28 |   26 | 0.5185 |  7.5000 |
| Devils Lake |   19 |   31 | 0.3800 | 14.5000 |
| Cavalier    |   15 |   32 | 0.3191 | 17.0000 |
+-------------+------+------+--------+---------+
```

There are a couple minor formatting issues to address at this point. Typically, standings listings display percentages to three decimal places, and the GB value to one decimal place (except that the GB value for the first-place team is displayed as `-`). To display *n* decimal places, use `TRUNCATE(`*expr*`,`*n*`)`. To display the GB value for the first-place team appropriately, use an `IF()` expression that maps 0 to a dash:

```
mysql> SELECT team, wins AS W, losses AS L,
    -> TRUNCATE(wins/(wins+losses),3) AS PCT,
    -> IF(@wl_diff = wins-losses,
    ->    '-',TRUNCATE((@wl_diff - (wins-losses))/2,1)) AS GB
    -> FROM standings1
```

```
    -> ORDER BY wins-losses DESC, PCT DESC;
+-------------+------+------+-------+------+
| team        | W    | L    | PCT   | GB   |
+-------------+------+------+-------+------+
| Winnipeg    |   37 |   20 | 0.649 | -    |
| Crookston   |   31 |   25 | 0.553 | 5.5  |
| Fargo       |   30 |   26 | 0.535 | 6.5  |
| Grand Forks |   28 |   26 | 0.518 | 7.5  |
| Devils Lake |   19 |   31 | 0.380 | 14.5 |
| Cavalier    |   15 |   32 | 0.319 | 17.0 |
+-------------+------+------+-------+------+
```

These statements order the teams by win-loss differential, using winning percentage as a tie-breaker in case there are teams with the same differential value. It's simpler to sort by percentage, of course, but then you wouldn't always get the correct ordering. It's a curious fact that a team with a lower winning percentage can actually be higher in the standings than a team with a higher percentage. (This generally occurs early in the season, when teams may have played highly disparate numbers of games, relatively speaking.) Consider the case in which two teams, A and B, have the following rows:

```
+------+------+--------+
| team | wins | losses |
+------+------+--------+
| A    |    4 |      1 |
| B    |    2 |      0 |
+------+------+--------+
```

Applying the GB and percentage calculations to these team records yields the following result, in which the first-place team actually has a lower winning percentage than the second-place team:

```
+------+------+------+-------+------+
| team | W    | L    | PCT   | GB   |
+------+------+------+-------+------+
| A    |    4 |    1 | 0.800 | -    |
| B    |    2 |    0 | 1.000 | 0.5  |
+------+------+------+-------+------+
```

The standings calculations shown thus far can be done without a join. They involve only a single set of team records, so the first-place team's win-loss differential can be stored in a variable. A more complex situation occurs when a dataset includes several sets of team records. For example, the 1997 Northern League had two divisions (Eastern and Western). In addition, separate standings were maintained for the first and second halves of the season because season-half winners in each division played each other for the right to compete in the league championship. The following table, `standings2`, shows what these rows look like, ordered by season half, division, and win-loss differential:

```
mysql> SELECT half, division, team, wins, losses FROM standings2
    -> ORDER BY half, division, wins-losses DESC;
```

```
+------+----------+-----------------+------+--------+
| half | division | team            | wins | losses |
+------+----------+-----------------+------+--------+
|    1 | Eastern  | St. Paul        |   24 |     18 |
|    1 | Eastern  | Thunder Bay     |   18 |     24 |
|    1 | Eastern  | Duluth-Superior |   17 |     24 |
|    1 | Eastern  | Madison         |   15 |     27 |
|    1 | Western  | Winnipeg        |   29 |     12 |
|    1 | Western  | Sioux City      |   28 |     14 |
|    1 | Western  | Fargo-Moorhead  |   21 |     21 |
|    1 | Western  | Sioux Falls     |   15 |     27 |
|    2 | Eastern  | Duluth-Superior |   22 |     20 |
|    2 | Eastern  | St. Paul        |   21 |     21 |
|    2 | Eastern  | Madison         |   19 |     23 |
|    2 | Eastern  | Thunder Bay     |   18 |     24 |
|    2 | Western  | Fargo-Moorhead  |   26 |     16 |
|    2 | Western  | Winnipeg        |   24 |     18 |
|    2 | Western  | Sioux City      |   22 |     20 |
|    2 | Western  | Sioux Falls     |   16 |     26 |
+------+----------+-----------------+------+--------+
```

Generating the standings for these rows requires computing the GB values separately for each of the four combinations of season half and division. First, calculate the win-loss differential for the first-place team in each group and save the values into a separate `firstplace` table:

```
mysql> CREATE TEMPORARY TABLE firstplace
    -> SELECT half, division, MAX(wins-losses) AS wl_diff
    -> FROM standings2
    -> GROUP BY half, division;
```

Then join the `firstplace` table to the original standings, associating each team record with the proper win-loss differential to compute its GB value:

```
mysql> SELECT wl.half, wl.division, wl.team, wl.wins AS W, wl.losses AS L,
    -> TRUNCATE(wl.wins/(wl.wins+wl.losses),3) AS PCT,
    -> IF(fp.wl_diff = wl.wins-wl.losses,
    ->    '-',TRUNCATE((fp.wl_diff - (wl.wins-wl.losses)) / 2,1)) AS GB
    -> FROM standings2 AS wl INNER JOIN firstplace AS fp
    -> ON wl.half = fp.half AND wl.division = fp.division
    -> ORDER BY wl.half, wl.division, wl.wins-wl.losses DESC, PCT DESC;
+------+----------+-----------------+------+------+-------+------+
| half | division | team            | W    | L    | PCT   | GB   |
+------+----------+-----------------+------+------+-------+------+
|    1 | Eastern  | St. Paul        |   24 |   18 | 0.571 | -    |
|    1 | Eastern  | Thunder Bay     |   18 |   24 | 0.428 | 6.0  |
|    1 | Eastern  | Duluth-Superior |   17 |   24 | 0.414 | 6.5  |
|    1 | Eastern  | Madison         |   15 |   27 | 0.357 | 9.0  |
|    1 | Western  | Winnipeg        |   29 |   12 | 0.707 | -    |
|    1 | Western  | Sioux City      |   28 |   14 | 0.666 | 1.5  |
|    1 | Western  | Fargo-Moorhead  |   21 |   21 | 0.500 | 8.5  |
|    1 | Western  | Sioux Falls     |   15 |   27 | 0.357 | 14.5 |
|    2 | Eastern  | Duluth-Superior |   22 |   20 | 0.523 | -    |
```

```
|    2 | Eastern  | St. Paul        |   21 |   21 | 0.500 | 1.0  |
|    2 | Eastern  | Madison         |   19 |   23 | 0.452 | 3.0  |
|    2 | Eastern  | Thunder Bay     |   18 |   24 | 0.428 | 4.0  |
|    2 | Western  | Fargo-Moorhead  |   26 |   16 | 0.619 | -    |
|    2 | Western  | Winnipeg        |   24 |   18 | 0.571 | 2.0  |
|    2 | Western  | Sioux City      |   22 |   20 | 0.523 | 4.0  |
|    2 | Western  | Sioux Falls     |   16 |   26 | 0.380 | 10.0 |
+------+----------+-----------------+------+------+-------+------+
```

That output is difficult to read, however. To make it easier to understand, you might execute the statement from within a program and reformat its results to display each set of team records separately. Here's some Perl code that does that by beginning a new output group each time it encounters a new group of standings. The code assumes that the join statement has just been executed and that its results are available through the statement handle `$sth`:

```
my ($cur_half, $cur_div) = ("", "");
while (my ($half, $div, $team, $wins, $losses, $pct, $gb)
          = $sth->fetchrow_array ())
{
  if ($cur_half ne $half || $cur_div ne $div) # new group of standings?
  {
    # print standings header and remember new half/division values
    print "\n$div Division, season half $half\n";
    printf "%-20s  %3s  %3s  %5s  %s\n", "Team", "W", "L", "PCT", "GB";
    $cur_half = $half;
    $cur_div = $div;
  }
  printf "%-20s  %3d  %3d  %5s  %s\n", $team, $wins, $losses, $pct, $gb;
}
```

The reformatted output looks like this:

```
Eastern Division, season half 1
Team                   W    L    PCT  GB
St. Paul              24   18  0.571  -
Thunder Bay           18   24  0.428  6.0
Duluth-Superior       17   24  0.414  6.5
Madison               15   27  0.357  9.0

Western Division, season half 1
Team                   W    L    PCT  GB
Winnipeg              29   12  0.707  -
Sioux City            28   14  0.666  1.5
Fargo-Moorhead        21   21  0.500  8.5
Sioux Falls           15   27  0.357  14.5

Eastern Division, season half 2
Team                   W    L    PCT  GB
Duluth-Superior       22   20  0.523  -
St. Paul              21   21  0.500  1.0
Madison               19   23  0.452  3.0
```

```
Thunder Bay            18   24  0.428  4.0

Western Division, season half 2
Team                    W    L    PCT  GB
Fargo-Moorhead         26   16  0.619  -
Winnipeg               24   18  0.571  2.0
Sioux City             22   20  0.523  4.0
Sioux Falls            16   26  0.380  10.0
```

The code just shown comes from the *calc_standings.pl* script in the *stats* directory of the `recipes` distribution. That directory also contains a PHP script, *calc_standings.php*, that produces output in the form of HTML tables, which you might prefer for generating standings in a web environment.

CHAPTER 16
Handling Duplicates

16.0. Introduction

Tables or result sets sometimes contain duplicate rows. In some cases this is acceptable. For example, if you conduct a web poll that records date and client IP number along with the votes, duplicate rows may be permitted because it's possible for large numbers of votes to appear to originate from the same IP number for an Internet service that routes traffic from its customers through a single proxy host. In other cases, duplicates are unacceptable, and you'll want to take steps to avoid them. Operations involved in handling duplicate rows include the following:

- Preventing duplicates from being created in the first place. If each row in a table is intended to represent a single entity (such as a person, an item in a catalog, or a specific observation in an experiment), the occurrence of duplicates presents significant difficulties in using it that way. Duplicates make it impossible to refer to each row unambiguously, so it's best to make sure duplicates never occur.
- Counting the number of duplicates to determine whether they are present and to what extent.
- Identifying duplicated values (or the rows containing them) so you can see where they occur.
- Eliminating duplicates to ensure that each row is unique. This may involve removing rows from a table to leave only unique rows or selecting a result set in such a way that no duplicates appear in the output. For example, to display a list of the states in which you have customers, you probably don't want a long list of state names from all customer records. A list showing each state name only once suffices and is easier to understand.

Several tools are at your disposal for dealing with duplicate rows. Choose them according to the objective that you want to achieve:

- When you create a table, include a primary key or unique index to prevent duplicates from being added to the table. MySQL uses the index as a constraint to enforce the requirement that each row in the table contains a unique key in the indexed column or columns.
- In conjunction with a unique index, the `INSERT IGNORE` and `REPLACE` statements enable you to handle insertion of duplicate rows gracefully without generating errors. For bulk-loading operations, the same options are available in the form of the `IGNORE` or `REPLACE` modifiers for the `LOAD DATA` statement.
- To determine whether a table contains duplicates, use `GROUP BY` to categorize rows into groups, and `COUNT()` to see how many rows are in each group. Chapter 8 describes these techniques in the context of producing summaries, but they're useful for duplicate counting and identification as well. A counting summary groups values into categories to determine how frequently each one occurs.
- `SELECT DISTINCT` removes duplicate rows from a result set (see Recipe 3.4 for more information). For an existing table that already contains duplicates, you can select unique rows into a second table and use it to replace the original table. Or, if you determine that there are *n* identical rows in a table, you can use `DELETE ... LIMIT` to eliminate *n*–1 instances from that specific set of rows.

Scripts related to the examples shown in this chapter are located in the *dups* directory of the `recipes` distribution. For scripts that create the tables used here, look in the *tables* directory.

16.1. Preventing Duplicates from Occurring in a Table

Problem

You want to prevent a table from ever containing duplicates.

Solution

Use a `PRIMARY KEY` or a `UNIQUE` index.

Discussion

To ensure that rows in a table are unique, some column or combination of columns must be required to contain unique values in each row. When this requirement is satisfied, you can refer to any row in the table unambiguously by using its unique identifier. To make sure a table has this characteristic, include a `PRIMARY KEY` or `UNIQUE` index in the table structure. The following table contains no such index, so it permits duplicate rows:

```
CREATE TABLE person
(
  last_name   CHAR(20),
  first_name  CHAR(20),
  address     CHAR(40)
);
```

To prevent multiple rows with the same first and last name values from being created in this table, add a `PRIMARY KEY` to its definition. When you do this, the indexed columns must be `NOT NULL`, because a `PRIMARY KEY` prohibits `NULL` values:

```
CREATE TABLE person
(
  last_name   CHAR(20) NOT NULL,
  first_name  CHAR(20) NOT NULL,
  address     CHAR(40),
  PRIMARY KEY (last_name, first_name)
);
```

The presence of a unique index in a table normally causes an error to occur if you insert a row into the table that duplicates an existing row in the column or columns that define the index. Recipe 16.2 discusses how to handle such errors or modify MySQL's duplicate-handling behavior.

Another way to enforce uniqueness is to add a `UNIQUE` index rather than a `PRIMARY KEY` to a table. The two types of indexes are similar, but a `UNIQUE` index can be created on columns that permit `NULL` values. For the `person` table, it's likely that you'd require both the first and last names to be filled in. If so, you still declare the columns as `NOT NULL`, and the following table definition is effectively equivalent to the preceding one:

```
CREATE TABLE person
(
  last_name   CHAR(20) NOT NULL,
  first_name  CHAR(20) NOT NULL,
  address     CHAR(40),
  UNIQUE (last_name, first_name)
);
```

If a `UNIQUE` index does happen to permit `NULL` values, `NULL` is special because it is the one value that can occur multiple times. The rationale for this is that it is not possible to know whether one unknown value is the same as another, so multiple unknown values are permitted.

Of course, you might want the `person` table to reflect the real world, in which people do sometimes have the same name. In this case, you cannot set up a unique index based on the name columns, because duplicate names must be permitted. Instead, each person must be assigned some sort of unique identifier, which becomes the value that distinguishes one row from another. In MySQL, it's common to accomplish this by using an `AUTO_INCREMENT` column:

```
CREATE TABLE person
(
  id          INT UNSIGNED NOT NULL AUTO_INCREMENT,
  last_name   CHAR(20),
  first_name  CHAR(20),
  address     CHAR(40),
  PRIMARY KEY (id)
);
```

In this case, when you create a row with an `id` value of `NULL`, MySQL assigns that column a unique ID automatically. Another possibility is to assign identifiers externally and use those IDs as unique keys. For example, citizens in a given country might have unique taxpayer ID numbers. If so, those numbers can serve as the basis for a unique index:

```
CREATE TABLE person
(
  tax_id      INT UNSIGNED NOT NULL,
  last_name   CHAR(20),
  first_name  CHAR(20),
  address     CHAR(40),
  PRIMARY KEY (tax_id)
);
```

See Also

If an existing table already contains duplicate rows that you want to remove, see Recipe 16.4. Chapter 13 further discusses `AUTO_INCREMENT` columns.

16.2. Dealing with Duplicates When Loading Rows into a Table

Problem

You've created a table with a unique index to prevent duplicate values in the indexed column or columns. But this results in an error if you attempt to insert a duplicate row, and you want to avoid having to deal with such errors.

Solution

One approach is to just ignore the error. Another is to use an `INSERT IGNORE`, `REPLACE`, or `INSERT ... ON DUPLICATE KEY UPDATE` statement, each of which modifies MySQL's duplicate-handling behavior. For bulk-loading operations, `LOAD DATA` has modifiers that enable you to specify how to handle duplicates.

Discussion

By default, MySQL generates an error when you insert a row that duplicates an existing unique key value. Suppose that the `person` table has the following structure, with a unique index on the `last_name` and `first_name` columns:

```
CREATE TABLE person
(
  last_name   CHAR(20) NOT NULL,
  first_name  CHAR(20) NOT NULL,
  address     CHAR(40),
  PRIMARY KEY (last_name, first_name)
);
```

An attempt to insert a row with duplicate values in the indexed columns results in an error:

```
mysql> INSERT INTO person (last_name, first_name)
    -> VALUES('X1','Y1');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO person (last_name, first_name)
    -> VALUES('X1','Y1');
ERROR 1062 (23000): Duplicate entry 'X1-Y1' for key 'PRIMARY'
```

If you issue the statements from the *mysql* program interactively, you can simply say, "Okay, that didn't work," ignore the error, and continue. But if you write a program to insert the rows, an error may terminate the program. One way to avoid this is to modify the program's error-handling behavior to trap the error and then ignore it. See Recipe 2.2 for information about error-handling techniques.

To prevent the error from occurring in the first place, you might consider using a two-query method to solve the duplicate-row problem:

- Issue a `SELECT` to check whether the row is already present.
- Issue an `INSERT` if the row is not present.

But that doesn't really work: another client might insert the same row after the `SELECT` and before the `INSERT`, in which case the error would still occur for your `INSERT`. To make sure that doesn't happen, you could use a transaction or lock the tables, but then you've gone from two statements to four. MySQL provides three single-query solutions to the problem of handling duplicate rows. Choose from among them depending on the duplicate-handling behavior you want:

- To keep the original row when a duplicate occurs, use `INSERT IGNORE` rather than `INSERT`. If the row duplicates no existing row, MySQL inserts it as usual. If the row is a duplicate, the `IGNORE` keyword tells MySQL to discard it silently without generating an error:

```
mysql> INSERT IGNORE INTO person (last_name, first_name)
    -> VALUES('X2','Y2');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT IGNORE INTO person (last_name, first_name)
    -> VALUES('X2','Y2');
Query OK, 0 rows affected (0.00 sec)
```

The row count value indicates whether the row was inserted or ignored. From within a program, you can obtain this value by checking the rows-affected function provided by your API (see Recipes 2.4 and 10.1).

- To replace the original row with the new one when a duplicate occurs, use `REPLACE` rather than `INSERT`. If the row is new, it's inserted just as with `INSERT`. If it's a duplicate, the new row replaces the old one:

```
mysql> REPLACE INTO person (last_name, first_name)
    -> VALUES('X3','Y3');
Query OK, 1 row affected (0.00 sec)
mysql> REPLACE INTO person (last_name, first_name)
    -> VALUES('X3','Y3');
Query OK, 2 rows affected (0.00 sec)
```

  The rows-affected value in the second case is 2 because the original row is deleted and the new row is inserted in its place.

- To modify columns of an existing row when a duplicate occurs, use `INSERT ... ON DUPLICATE KEY UPDATE`. If the row is new, it's inserted. If it's a duplicate, the `ON DUPLICATE KEY UPDATE` clause indicates how to modify the existing row in the table. In other words, this statement can insert or update a row as necessary. The rows-affected count indicates what happened: 1 for an insert, 2 for an update.

`INSERT IGNORE` is more efficient than `REPLACE` because it doesn't actually insert duplicates. Thus, it's most applicable when you just want to make sure a copy of a given row is present in a table. `REPLACE`, on the other hand, is often more appropriate for tables in which other nonkey columns need to be replaced. `INSERT ... ON DUPLICATE KEY UPDATE` is appropriate when you must insert a record if it doesn't exist, but just update some of its columns if the new record is a duplicate in the indexed columns.

Suppose that you maintain a table named `passtbl` for a web application that contains email addresses and password hash values, and that is indexed by email address:

```
CREATE TABLE passtbl
(
  email    VARCHAR(60) NOT NULL,
  password VARBINARY(60) NOT NULL,
  PRIMARY KEY (email)
);
```

How do you create new rows for new users, but change passwords of existing rows for existing users? Here's a typical algorithm for handling row maintenance:

1. Issue a SELECT to check whether a row already exists with a given email value.
2. If no such row exists, add a new one with INSERT.
3. If the row does exist, update it with UPDATE.

These steps must be performed within a transaction or with the tables locked to prevent other users from changing the tables while you're using them. In MySQL, you can use REPLACE to simplify both cases to the same single-statement operation:

```
REPLACE INTO passtbl (email,password) VALUES(address,hash_value);
```

If no row with the given email address exists, MySQL creates a new one. Otherwise, MySQL replaces it, in effect updating the password column of the row associated with the address.

INSERT IGNORE and REPLACE are useful when you know exactly what values should be stored in the table when you attempt to insert a row. That's not always the case. For example, you might want to insert a row if it doesn't exist, but update only certain parts of it otherwise. This commonly occurs when you use a table for counting. Suppose that you record votes for candidates in polls, using the following table:

```
CREATE TABLE poll_vote
(
  poll_id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
  candidate_id INT UNSIGNED,
  vote_count   INT UNSIGNED,
  PRIMARY KEY (poll_id, candidate_id)
);
```

The primary key is the combination of poll and candidate number. The table should be used like this:

- For the first vote received for a given poll candidate, insert a new row with a vote count of 1.
- For subsequent votes for that candidate, increment the vote count of the existing record.

Neither INSERT IGNORE nor REPLACE are appropriate here because for all votes except the first, you don't know what the vote count should be. INSERT … ON DUPLICATE KEY UPDATE works better here. The following example shows how it works, beginning with an empty table:

```
mysql> SELECT * FROM poll_vote;
Empty set (0.00 sec)
mysql> INSERT INTO poll_vote (poll_id,candidate_id,vote_count) VALUES(14,3,1)
    -> ON DUPLICATE KEY UPDATE vote_count = vote_count + 1;
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM poll_vote;
```

```
+---------+-------------+------------+
| poll_id | candidate_id | vote_count |
+---------+-------------+------------+
|      14 |            3 |          1 |
+---------+-------------+------------+
1 row in set (0.00 sec)
mysql> INSERT INTO poll_vote (poll_id,candidate_id,vote_count) VALUES(14,3,1)
    -> ON DUPLICATE KEY UPDATE vote_count = vote_count + 1;
Query OK, 2 rows affected (0.00 sec)
mysql> SELECT * FROM poll_vote;
+---------+-------------+------------+
| poll_id | candidate_id | vote_count |
+---------+-------------+------------+
|      14 |            3 |          2 |
+---------+-------------+------------+
1 row in set (0.00 sec)
```

For the first `INSERT`, no row for the candidate exists, so the row is inserted. For the second `INSERT`, the row exists, so MySQL just updates the vote count. With `INSERT … ON DUPLICATE KEY UPDATE`, you need not check whether the row exists; MySQL does it for you. The row count indicates what action the `INSERT` statement performs: 1 for a new row and 2 for an update to an existing row.

The techniques just described have the benefit of eliminating overhead that might otherwise be required for a transaction. But this benefit comes at the price of portability because they all involve MySQL-specific syntax. If portability is a high priority, you might prefer to use a transactional approach.

See Also

For bulk record-loading operations in which you use the `LOAD DATA` statement to load a set of rows from a file into a table, control duplicate-row handling using the statement's `IGNORE` and `REPLACE` modifiers. These produce behavior analogous to that of the `INSERT IGNORE` and `REPLACE` statements. For more information, see Recipe 11.1.

Recipes 13.12 and 20.12 further demonstrate the use of `INSERT … ON DUPLICATE KEY UPDATE` for initializing and updating counts.

16.3. Counting and Identifying Duplicates

Problem

You want to determine whether a table contains duplicates, and to what extent they occur. Or you want to see the rows that contain the duplicated values.

Solution

Use a counting summary that displays duplicated values. To see the rows in which the duplicated values occur, join the summary to the original table to display the matching rows.

Discussion

Suppose that your website has a sign-up page that enables visitors to add themselves to your mailing list to receive periodic product catalog mailings. But you forgot to include a unique index in the table when you created it, and now you suspect that some people are signed up multiple times. Perhaps they forgot they were already on the list, or perhaps people added friends to the list who were already signed up. Either way, the result of having duplicate rows is that you mail out duplicate catalogs. This is an additional expense to you, and it annoys the recipients. This section discusses how to determine whether there are duplicate rows in a table, how prevalent they are, and how to display them. (For tables that do contain duplicates, Recipe 16.4 describes how to eliminate them.)

To determine whether duplicates occur in a table, use a counting summary (a topic covered in Chapter 8). Summary techniques can be applied to identifying and counting duplicates by grouping rows with `GROUP BY` and counting the rows in each group using `COUNT()`. For the examples here, assume that catalog recipients are listed in a table named `catalog_list` that has the following contents:

```
+-----------+------------+-------------------------+
| last_name | first_name | street                  |
+-----------+------------+-------------------------+
| Isaacson  | Jim        | 515 Fordam St., Apt. 917 |
| Baxter    | Wallace    | 57 3rd Ave.             |
| McTavish  | Taylor     | 432 River Run           |
| Pinter    | Marlene    | 9 Sunset Trail          |
| BAXTER    | WALLACE    | 57 3rd Ave.             |
| Brown     | Bartholomew | 432 River Run          |
| Pinter    | Marlene    | 9 Sunset Trail          |
| Baxter    | Wallace    | 57 3rd Ave., Apt 102    |
+-----------+------------+-------------------------+
```

Suppose that you define "duplicate" using the `last_name` and `first_name` columns. That is, recipients with the same name are assumed to be the same person. The following statements characterize the table and assess the existence and extent of duplicate values:

- The total number of rows in the table:

```
mysql> SELECT COUNT(*) AS rows FROM catalog_list;
+------+
| rows |
+------+
```

```
|    8 |
+------+
```

- The number of distinct names:

```
mysql> SELECT COUNT(DISTINCT last_name, first_name) AS 'distinct names'
    -> FROM catalog_list;
+----------------+
| distinct names |
+----------------+
|              5 |
+----------------+
```

- The number of rows containing duplicated names:

```
mysql> SELECT COUNT(*) - COUNT(DISTINCT last_name, first_name)
    ->   AS 'duplicate names'
    -> FROM catalog_list;
+-----------------+
| duplicate names |
+-----------------+
|               3 |
+-----------------+
```

- The fraction of the rows that contain unique or nonunique names:

```
mysql> SELECT COUNT(DISTINCT last_name, first_name) / COUNT(*)
    ->   AS 'unique',
    -> 1 - (COUNT(DISTINCT last_name, first_name) / COUNT(*))
    ->   AS 'nonunique'
    -> FROM catalog_list;
+--------+-----------+
| unique | nonunique |
+--------+-----------+
| 0.6250 |    0.3750 |
+--------+-----------+
```

Those statements help you characterize the extent of duplicates, but they don't show you which values are duplicated. To see the duplicated names in the `catalog_list` table, use a summary statement that displays the nonunique values along with the counts:

```
mysql> SELECT COUNT(*), last_name, first_name
    -> FROM catalog_list
    -> GROUP BY last_name, first_name
    -> HAVING COUNT(*) > 1;
+----------+-----------+------------+
| COUNT(*) | last_name | first_name |
+----------+-----------+------------+
|        3 | Baxter    | Wallace    |
|        2 | Pinter    | Marlene    |
+----------+-----------+------------+
```

The statement includes a HAVING clause that restricts the output to include only those names that occur more than once. In general, to identify sets of values that are duplicated, do the following:

1. Determine which columns contain the values that may be duplicated.
2. List those columns in the column selection list, along with COUNT(*).
3. List the columns in the GROUP BY clause as well.
4. Add a HAVING clause that eliminates unique values by requiring group counts to be greater than one.

Queries constructed that way have the following form:

```
SELECT COUNT(*), column_list
FROM tbl_name
GROUP BY column_list
HAVING COUNT(*) > 1
```

It's easy to generate duplicate-finding queries like that within a program, given database and table names and a nonempty set of column names. For example, here is a Perl function make_dup_count_query() that generates the proper query for finding and counting duplicated values in the specified columns:

```
sub make_dup_count_query
{
my ($db_name, $tbl_name, @col_name) = @_;

  return "SELECT COUNT(*)," . join (",", @col_name)
         . "\nFROM $db_name.$tbl_name"
         . "\nGROUP BY " . join (",", @col_name)
         . "\nHAVING COUNT(*) > 1";
}
```

make_dup_count_query() returns the query as a string. If you invoke it like this:

```
$str = make_dup_count_query ("cookbook", "catalog_list",
                             "last_name", "first_name");
```

the resulting value of $str is:

```
SELECT COUNT(*),last_name,first_name
FROM cookbook.catalog_list
GROUP BY last_name,first_name
HAVING COUNT(*) > 1
```

What you do with the query string is up to you. You can execute it from within the script that creates it, pass it to another program, or write it to a file for execution later. The *dups* directory of the recipes distribution contains a script named *dup_count.pl* that you can use to try the function (as well as some translations into other languages).

Recipe 16.4 discusses use of `make_dup_count_query()` to implement a duplicate-removal technique.

Summary techniques are useful for assessing the existence of duplicates, how often they occur, and displaying which values are duplicated. But if duplicates are determined using only a subset of a table's columns, a summary in itself cannot display the entire content of the rows that contain the duplicate values. (For example, the summaries shown thus far display counts of duplicated names in the `catalog_list` table or the names themselves, but don't show the addresses associated with those names.) To see the original rows containing the duplicate names, join the summary information to the table from which it's generated. The following example shows how to do this to display the `catalog_list` rows that contain duplicated names. The summary is written to a temporary table, which then is joined to the `catalog_list` table to produce the rows that match those names:

```
mysql> CREATE TABLE tmp
    -> SELECT COUNT(*) AS count, last_name, first_name FROM catalog_list
    -> GROUP BY last_name, first_name HAVING count > 1;
mysql> SELECT catalog_list.*
    -> FROM tmp INNER JOIN catalog_list USING (last_name, first_name)
    -> ORDER BY last_name, first_name;
+-----------+------------+----------------------+
| last_name | first_name | street               |
+-----------+------------+----------------------+
| Baxter    | Wallace    | 57 3rd Ave.          |
| BAXTER    | WALLACE    | 57 3rd Ave.          |
| Baxter    | Wallace    | 57 3rd Ave., Apt 102 |
| Pinter    | Marlene    | 9 Sunset Trail       |
| Pinter    | Marlene    | 9 Sunset Trail       |
+-----------+------------+----------------------+
```

Duplicate Identification and String Case Sensitivity

For strings that have a case-insensitive collation, values that differ only in lettercase are considered the same for comparison purposes. To treat them as distinct values, compare them using a case-sensitive or binary collation. Recipe 5.7 shows how to do this.

16.4. Eliminating Duplicates from a Table

Problem

You want to remove duplicate rows from a table, leaving only unique rows.

Solution

Select the unique rows from the table into a second table, then use it to replace the original one. Or use `DELETE` … `LIMIT` *n* to remove all but one instance of a specific set of duplicate rows.

Discussion

Recipe 16.1 discusses how to prevent duplicates from being added to a table by creating it with a unique index. However, if you forget to include the index when you create a table, you may discover later that it contains duplicates and that it's necessary to apply some sort of duplicate-removal technique. The `catalog_list` table used earlier is an example of this because it contains several instances in which the same person appears multiple times:

```
mysql> SELECT * FROM catalog_list ORDER BY last_name, first_name;
+-----------+------------+-------------------------+
| last_name | first_name | street                  |
+-----------+------------+-------------------------+
| Baxter    | Wallace    | 57 3rd Ave.             |
| BAXTER    | WALLACE    | 57 3rd Ave.             |
| Baxter    | Wallace    | 57 3rd Ave., Apt 102    |
| Brown     | Bartholomew | 432 River Run          |
| Isaacson  | Jim        | 515 Fordam St., Apt. 917 |
| McTavish  | Taylor     | 432 River Run           |
| Pinter    | Marlene    | 9 Sunset Trail          |
| Pinter    | Marlene    | 9 Sunset Trail          |
+-----------+------------+-------------------------+
```

To eliminate duplicates, you have a few options:

- Select the table's unique rows into another table, then use that table to replace the original one. This works when "duplicate" means "the entire row is the same as another."
- To remove duplicates for a specific set of duplicate rows, use `DELETE` … `LIMIT` *n* to remove all but one row.

This recipe discusses each duplicate-removal method. When you consider which to choose under various circumstances, the applicability of a given method to a specific problem is often determined by several factors:

- Does the method require the table to have a unique index?
- If the columns in which duplicate values occur may contain `NULL`, will the method remove duplicate `NULL` values?
- Does the method prevent duplicates from occurring in the future?

Removing duplicates using table replacement

If a row is considered to duplicate another only if the entire row is the same, one way to eliminate duplicates from a table is to select its unique rows into a new table that has the same structure, and then replace the original table with the new one:

1. Create a new table that has the same structure as the original one. `CREATE TABLE ... LIKE` is useful for this (see Recipe 4.1):

   ```
   mysql> CREATE TABLE tmp LIKE catalog_list;
   ```

2. Use `INSERT INTO ... SELECT DISTINCT` to select the unique rows from the original table into the new one:

   ```
   mysql> INSERT INTO tmp SELECT DISTINCT * FROM catalog_list;
   ```

   Select rows from the `tmp` table to verify that the new table contains no duplicates:

   ```
   mysql> SELECT * FROM tmp ORDER BY last_name, first_name;
   +-----------+-------------+-------------------------+
   | last_name | first_name  | street                  |
   +-----------+-------------+-------------------------+
   | Baxter    | Wallace     | 57 3rd Ave.             |
   | Baxter    | Wallace     | 57 3rd Ave., Apt 102    |
   | Brown     | Bartholomew | 432 River Run           |
   | Isaacson  | Jim         | 515 Fordam St., Apt. 917 |
   | McTavish  | Taylor      | 432 River Run           |
   | Pinter    | Marlene     | 9 Sunset Trail          |
   +-----------+-------------+-------------------------+
   ```

3. After creating the new `tmp` table that contains unique rows, use it to replace the original `catalog_list` table:

   ```
   mysql> DROP TABLE catalog_list;
   mysql> RENAME TABLE tmp TO catalog_list;
   ```

The effective result of this procedure is that `catalog_list` no longer contains duplicates.

This table-replacement method works in the absence of an index (although it might be slow for large tables). For tables that contain duplicate `NULL` values, it removes those duplicates. It does not prevent the occurrence of duplicates in the future.

This method requires rows to be completely identical to be considered duplicates. Thus, it treats as distinct those rows for Wallace Baxter that have slightly different `street` values.

If duplicates are defined only with respect to a subset of the columns in the table, create a new table that has a unique index for those columns, select rows into it using `INSERT IGNORE`, and replace the original table with the new one:

```
mysql> CREATE TABLE tmp LIKE catalog_list;
mysql> ALTER TABLE tmp ADD PRIMARY KEY (last_name, first_name);
mysql> INSERT IGNORE INTO tmp SELECT * FROM catalog_list;
```

```
mysql> SELECT * FROM tmp ORDER BY last_name, first_name;
+-----------+-------------+--------------------------+
| last_name | first_name  | street                   |
+-----------+-------------+--------------------------+
| Baxter    | Wallace     | 57 3rd Ave.              |
| Brown     | Bartholomew | 432 River Run            |
| Isaacson  | Jim         | 515 Fordam St., Apt. 917 |
| McTavish  | Taylor      | 432 River Run            |
| Pinter    | Marlene     | 9 Sunset Trail           |
+-----------+-------------+--------------------------+
mysql> DROP TABLE catalog_list;
mysql> RENAME TABLE tmp TO catalog_list;
```

The unique index prevents rows with duplicate key values from being inserted into `tmp`, and `IGNORE` tells MySQL not to stop with an error if a duplicate is found. One shortcoming of this method is that if the indexed columns can contain `NULL` values, you must use a `UNIQUE` index rather than a `PRIMARY KEY`, in which case the index will not remove duplicate `NULL` keys. (`UNIQUE` indexes permit multiple `NULL` values.) This method does prevent occurrence of duplicates in the future.

Removing duplicates of a particular row

You can use `LIMIT` to restrict the effect of a `DELETE` statement to a subset of the rows that it otherwise would delete. This makes the statement applicable to removing duplicate rows. Suppose that the original unindexed `catalog_list` table contains duplicates:

```
mysql> SELECT COUNT(*), last_name, first_name
    -> FROM catalog_list
    -> GROUP BY last_name, first_name
    -> HAVING COUNT(*) > 1;
+----------+-----------+------------+
| COUNT(*) | last_name | first_name |
+----------+-----------+------------+
|        3 | Baxter    | Wallace    |
|        2 | Pinter    | Marlene    |
+----------+-----------+------------+
```

To remove the extra instances of each name, do this:

```
mysql> DELETE FROM catalog_list WHERE last_name = 'Baxter'
    -> AND first_name = 'Wallace' LIMIT 2;
mysql> DELETE FROM catalog_list WHERE last_name = 'Pinter'
    -> AND first_name = 'Marlene' LIMIT 1;
mysql> SELECT * FROM catalog_list;
+-----------+-------------+--------------------------+
| last_name | first_name  | street                   |
+-----------+-------------+--------------------------+
| Isaacson  | Jim         | 515 Fordam St., Apt. 917 |
| McTavish  | Taylor      | 432 River Run            |
| Brown     | Bartholomew | 432 River Run            |
| Pinter    | Marlene     | 9 Sunset Trail           |
```

```
| Baxter    | Wallace     | 57 3rd Ave., Apt 102     |
+-----------+-------------+--------------------------+
```

This technique works in the absence of a unique index, and it eliminates duplicate `NULL` values. It's handy for removing duplicates only for a specific set of rows within a table. However, if there are many different sets of duplicates to remove, this is not a procedure you'd want to carry out by hand. The process can be automated by using the techniques discussed earlier in Recipe 16.3 for determining which values are duplicated. There, we wrote a `make_dup_count_query()` function to generate the statement needed to count the number of duplicate values in a given set of columns in a table. The result of that statement can be used to generate a set of `DELETE … LIMIT` *n* statements that remove duplicate rows and leave only unique rows. The *dups* directory of the `recipes` distribution contains code that shows how to generate these statements.

In general, using `DELETE … LIMIT` *n* is likely to be slower than removing duplicates by using a second table or by adding a unique index. Those methods keep the data on the server side and let the server do all the work. `DELETE … LIMIT` *n* involves a lot of client-server interaction because it uses a `SELECT` statement to retrieve information about duplicates, followed by several `DELETE` statements to remove instances of duplicated rows. Also, this technique does not prevent duplicates from occurring in the future.

CHAPTER 17
Performing Transactions

17.0. Introduction

The MySQL server can handle multiple clients at the same time because it is multi-threaded. To deal with contention among clients, the server performs any necessary locking so that two clients cannot modify the same data at once. However, as the server executes SQL statements, it's very possible that successive statements received from a given client will be interleaved with statements from other clients. If a client executes multiple statements that are dependent on each other, the fact that other clients may be updating tables in between those statements can cause difficulties. Statement failures can be problematic, too, if a multiple-statement operation does not run to completion. Suppose that a `flight` table contains information about airline flight schedules and you want to update the row for Flight 578 by choosing a pilot from among those available. You might do so using three statements as follows:

```
SET @p_val = (SELECT pilot_id FROM pilot WHERE available = 'yes' LIMIT 1);
UPDATE pilot SET available = 'no' WHERE pilot_id = @p_val;
UPDATE flight SET pilot_id = @p_val WHERE flight_id = 578;
```

The first statement chooses an available pilot, the second marks the pilot as unavailable, and the third assigns the pilot to the flight. That's straightforward enough in principle, but in practice there are significant difficulties:

Concurrency issues
: If two clients want to schedule pilots, it's possible for both to run the initial `SELECT` query and retrieve the same pilot ID number before either has a chance to set the pilot's status to unavailable. If that happens, the same pilot is scheduled for two flights at once.

Integrity issues
: All three statements must execute successfully as a unit. For example, if the `SELECT` and the first `UPDATE` run successfully, but the second `UPDATE` fails, the pilot's

status is set to unavailable without the pilot being assigned a flight. The database becomes inconsistent.

To prevent concurrency and integrity problems in these types of situations, transactions are helpful. A transaction groups a set of statements and guarantees the following properties:

- No other client can update the data used in the transaction while the transaction is in progress; it's as though you have the server all to yourself. For example, other clients cannot modify the pilot or flight records while you're booking a pilot for a flight. Transactions solve concurrency problems arising from the multiple-client nature of the MySQL server. In effect, transactions serialize access to a shared resource across multiple-statement operations.
- Statements grouped within a transaction are committed (take effect) as a unit, but only if they all succeed. If an error occurs, any actions that occurred prior to the error are rolled back, leaving the relevant tables unaffected as though none of the statements had been executed. This keeps the database from becoming inconsistent. For example, if an update to the `flights` table fails, rollback causes the change to the `pilots` table to be undone, leaving the pilot still available. Rollback frees you from having to figure out how to undo a partially completed operation yourself.

This chapter shows the syntax for the SQL statements that begin and end transactions. It also describes how to implement transactional operations from within programs, using error detection to determine whether to commit or roll back.

Scripts related to the examples shown here are located in the *transactions* directory of the `recipes` distribution.

17.1. Choosing a Transactional Storage Engine

Problem

You want to use transactions.

Solution

Check your MySQL server to determine which transactional storage engines it supports.

Discussion

MySQL supports several storage engines, but to use transactions, you must use a transaction-safe engine. Currently, the transactional engines include InnoDB and NDB. To see which your MySQL server supports, use this statement:

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.ENGINES
    -> WHERE SUPPORT IN ('YES','DEFAULT') AND TRANSACTIONS='YES';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+
```

If MySQL Cluster is enabled, you'll also see a line that says `ndbcluster`.

Transactional engines are those that have a `TRANSACTIONS` value of `YES`; those actually usable have a `SUPPORT` value of `YES` or `DEFAULT`.

After determining which transactional storage engines are available, to create a table that uses a given engine, add an `ENGINE = tbl_engine` clause to your `CREATE TABLE` statement:

```
CREATE TABLE t (i INT) ENGINE = InnoDB;
```

If you need to modify an existing application to perform transactions, but it uses nontransactional tables, you can alter the tables to use a transactional storage engine. For example, MyISAM tables are nontransactional and trying to use them for transactions will yield incorrect results because they do not support rollback. In this case, you can use `ALTER TABLE` to convert the tables to a transactional type. Suppose that `t` is a MyISAM table. To make it an InnoDB table, do this:

```
ALTER TABLE t ENGINE = InnoDB;
```

One thing to consider before altering a table is that changing it to use a transactional storage engine may affect its behavior in other ways. For example, the MyISAM engine provides more flexible handling of `AUTO_INCREMENT` columns than do other storage engines. If you rely on MyISAM-only sequence features, changing the storage engine will cause problems.

17.2. Performing Transactions Using SQL

Problem

A set of statements must succeed or fail as a unit—that is, you require a transaction.

Solution

Manipulate MySQL's auto-commit mode to enable multiple-statement transactions, and then commit or roll back the statements depending on whether they succeed or fail.

Discussion

This recipe describes the SQL statements that control transactional behavior in MySQL. The immediately following recipes discuss how to perform transactions from within programs. Some APIs require that you implement transactions by executing the SQL statements discussed in this recipe; others provide a special mechanism that enables transaction management without writing SQL directly. However, even in the latter case, the API mechanism maps program operations onto transactional SQL statements, so reading this recipe will give you a better understanding of what the API does on your behalf.

MySQL normally operates in auto-commit mode, which commits the effect of each statement as soon as it executes. (In effect, each statement is its own transaction.) To perform a transaction, you must disable auto-commit mode, execute the statements that make up the transaction, and then either commit or roll back your changes. In MySQL, you can do this two ways:

- Execute a `START TRANSACTION` (or `BEGIN`) statement to suspend auto-commit mode, then execute the statements that make up the transaction. If the statements succeed, record their effect in the database and terminate the transaction by executing a `COMMIT` statement:

```
mysql> CREATE TABLE t (i INT) ENGINE = InnoDB;
mysql> START TRANSACTION;
mysql> INSERT INTO t (i) VALUES(1);
mysql> INSERT INTO t (i) VALUES(2);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| i    |
+------+
|    1 |
|    2 |
+------+
```

  If an error occurs, don't use `COMMIT`. Instead, cancel the transaction by executing a `ROLLBACK` statement. In the following example, `t` remains empty after the transaction because the effects of the `INSERT` statements are rolled back:

```
mysql> CREATE TABLE t (i INT) ENGINE = InnoDB;
mysql> START TRANSACTION;
mysql> INSERT INTO t (i) VALUES(1);
mysql> INSERT INTO t (x) VALUES(2);
ERROR 1054 (42S22): Unknown column 'x' in 'field list'
mysql> ROLLBACK;
mysql> SELECT * FROM t;
Empty set (0.00 sec)
```

- Another way to group statements is to turn off auto-commit mode explicitly by setting the `autocommit` session variable to 0. After that, each statement you execute

becomes part of the current transaction. To end the transaction and begin the next one, execute a COMMIT or ROLLBACK statement:

```
mysql> CREATE TABLE t (i INT) ENGINE = InnoDB;
mysql> SET autocommit = 0;
mysql> INSERT INTO t (i) VALUES(1);
mysql> INSERT INTO t (i) VALUES(2);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| i    |
+------+
|    1 |
|    2 |
+------+
```

To turn auto-commit mode back on, use this statement:

```
mysql> SET autocommit = 1;
```

Transactions have their limits because not all statements can be part of a transaction. For example, if you execute a DROP DATABASE statement, don't expect to restore the database by executing a ROLLBACK.

17.3. Performing Transactions from Within Programs

Problem

You're writing a program that must implement transactional operations.

Solution

Use the transaction abstraction provided by your language API, if it has such a thing. If it doesn't, use the API's usual statement-execution mechanism to execute the transactional SQL statements directly.

Discussion

To perform transactional processing from within a program, use your API language to detect errors and take appropriate action. This recipe provides general background on doing this. The next recipes provide language-specific details for the MySQL APIs for Perl, Ruby, PHP, Python, and Java.

Every MySQL API supports transactions, even if only in the sense that you can explicitly execute transaction-related SQL statements such as START TRANSACTION and COMMIT.

However, some APIs also provide a transaction abstraction that enables control over transactional behavior without working directly with SQL. That approach hides the details and provides better portability to other database engines that have different underlying transaction SQL syntax. An API abstraction is available for each language that we use in this book.

The next few recipes each implement the same example to illustrate how to perform program-based transactions. They use a table `t` containing the following initial rows that show how much money two people have:

```
+------+------+
| name | amt  |
+------+------+
| Eve  |   10 |
| Ida  |    0 |
+------+------+
```

The sample transaction is a simple financial transfer that uses two `UPDATE` statements to give six dollars of Eve's money to Ida:

```
UPDATE money SET amt = amt - 6 WHERE name = 'Eve';
UPDATE money SET amt = amt + 6 WHERE name = 'Ida';
```

The intended result is that the table should look like this:

```
+------+------+
| name | amt  |
+------+------+
| Eve  |    4 |
| Ida  |    6 |
+------+------+
```

It's necessary to execute both statements within a transaction to ensure that both of them take effect at once. Without a transaction, Eve's money disappears without being credited to Ida if the second statement fails. By using a transaction, the table is left unchanged if statement failure occurs.

The sample programs for each language are located in the *transactions* directory of the `recipes` distribution. If you compare them, you'll see that they all employ a similar framework for performing transactional processing:

- The transaction statements are grouped within a control structure, along with a commit operation.
- If the status of the control structure indicates that it did not execute successfully to completion, the transaction is rolled back.

That logic can be expressed as follows, where `block` represents the control structure used to group statements:

```
block:
  statement 1
  statement 2
  ...
  statement n
  commit
if the block failed:
  roll back
```

If the statements in the block succeed, you reach the end of the block and perform a commit. Otherwise, occurrence of an error raises an exception that triggers execution of the error-handling code where you roll back the transaction.

The benefit of structuring your code as just described is that it minimizes the number of tests needed to determine whether to roll back. The alternative—checking the result of each statement within the transaction and rolling back on individual statement errors—quickly turns your code into an unreadable mess.

A subtle point to be aware of when rolling back within languages that raise exceptions is that it may be possible for the rollback itself to fail, causing another exception to be raised. If you don't deal with that, your program itself may terminate. To handle this, execute the rollback within another block that has an empty exception handler. The sample programs do this as necessary.

Those sample programs that disable auto-commit mode explicitly when performing a transaction enable auto-commit afterward. In applications that perform all database processing in transactional fashion, it's unnecessary to do this. Just disable auto-commit mode once after you connect to the database server, and leave it off.

Checking How API Transaction Abstractions Map onto SQL Statements

For APIs that provide a transaction abstraction, you can see how the interface maps onto the underlying SQL statements: enable the general query log for your MySQL server, then watch the log to see what statements the API executes when you run a transactional program. For instructions on enabling the log, see Recipe 22.3.

17.4. Using Transactions in Perl Programs

Problem

You want to perform a transaction in a Perl DBI script.

Solution

Use the standard DBI transaction support mechanism.

Discussion

The Perl DBI transaction mechanism is based on explicit manipulation of auto-commit mode:

1. Turn on the `RaiseError` attribute if it's not enabled and disable `PrintError` if it's on. You want errors to raise exceptions without printing anything, and leaving `PrintError` enabled can interfere with failure detection in some cases.
2. Disable the `AutoCommit` attribute so that a commit will be done only when you say so.
3. Execute the statements that make up the transaction within an `eval` block so that errors raise an exception and terminate the block. The last thing in the block should be a call to `commit()`, which commits the transaction if all its statements completed successfully.
4. After the `eval` executes, check the `$@` variable. If `$@` contains the empty string, the transaction succeeded. Otherwise, the `eval` will have failed due to the occurrence of some error and `$@` will contain an error message. Invoke `rollback()` to cancel the transaction. To display an error message, print `$@` before calling `rollback()`.
5. If desired, restore the original values of the `RaiseError` and `PrintError` attributes.

Because it can be messy to change and restore the error-handling and auto-commit attributes if an application performs multiple transactions, let's put the code to begin and end a transaction into convenience functions that handle the processing that occurs before and after the `eval`:

```
sub transaction_init
{
my $dbh = shift;
my $attr_ref = {};  # create hash in which to save attributes

  $attr_ref->{RaiseError} = $dbh->{RaiseError};
  $attr_ref->{PrintError} = $dbh->{PrintError};
  $attr_ref->{AutoCommit} = $dbh->{AutoCommit};
  $dbh->{RaiseError} = 1; # raise exception if an error occurs
  $dbh->{PrintError} = 0; # don't print an error message
  $dbh->{AutoCommit} = 0; # disable auto-commit
  return $attr_ref;       # return attributes to caller
}

sub transaction_finish
{
my ($dbh, $attr_ref, $error) = @_;

  if ($error) # an error occurred
  {
    print "Transaction failed, rolling back. Error was:\n$error\n";
```

```
    # roll back within eval to prevent rollback
    # failure from terminating the script
    eval { $dbh->rollback (); };
  }
  # restore error-handling and auto-commit attributes
  $dbh->{AutoCommit} = $attr_ref->{AutoCommit};
  $dbh->{PrintError} = $attr_ref->{PrintError};
  $dbh->{RaiseError} = $attr_ref->{RaiseError};
}
```

By using those two functions, our sample transaction can be performed easily as follows:

```
$ref = transaction_init ($dbh);
eval
{
  # move some money from one person to the other
  $dbh->do ("UPDATE money SET amt = amt - 6 WHERE name = 'Eve'");
  $dbh->do ("UPDATE money SET amt = amt + 6 WHERE name = 'Ida'");
  # all statements succeeded; commit transaction
  $dbh->commit ();
};
transaction_finish ($dbh, $ref, $@);
```

In Perl DBI, an alternative to manipulating the `AutoCommit` attribute manually is to begin a transaction by invoking `begin_work()`. This method disables `AutoCommit` and causes it to be enabled again automatically when you invoke `commit()` or `rollback()` later.

17.5. Using Transactions in Ruby Programs

Problem

You want to perform a transaction in a Ruby DBI script.

Solution

Use the standard DBI transaction support mechanism. Actually, Ruby provides two mechanisms.

Discussion

The Ruby DBI module provides two ways to perform transactions, although both of them rely on manipulation of auto-commit mode. One approach uses a `begin/rescue` block, and you invoke the `commit` and `rollback` methods explicitly:

```
begin
  dbh['AutoCommit'] = false
  dbh.do("UPDATE money SET amt = amt - 6 WHERE name = 'Eve'")
  dbh.do("UPDATE money SET amt = amt + 6 WHERE name = 'Ida'")
```

```
    dbh.commit
    dbh['AutoCommit'] = true
  rescue DBI::DatabaseError => e
    puts "Transaction failed, rolling back. Error was:"
    puts "#{e.err}: #{e.errstr}"
    begin           # empty exception handler in case rollback fails
      dbh.rollback
      dbh['AutoCommit'] = true
    rescue
    end
  end
```

Ruby also supports a `transaction` method, which is associated with a code block and commits or rolls back automatically depending on whether the code block succeeds or fails:

```
  begin
    dbh['AutoCommit'] = false
    dbh.transaction do |dbh|
      dbh.do("UPDATE money SET amt = amt - 6 WHERE name = 'Eve'")
      dbh.do("UPDATE money SET amt = amt + 6 WHERE name = 'Ida'")
    end
    dbh['AutoCommit'] = true
  rescue DBI::DatabaseError => e
    puts "Transaction failed, rolling back. Error was:"
    puts "#{e.err}: #{e.errstr}"
    dbh['AutoCommit'] = true
  end
```

With the `transaction` method, there is no need to invoke `commit` or `rollback` explicitly. `transaction` does raise an exception if it rolls back, so the example still uses a `begin/rescue` block for error detection.

17.6. Using Transactions in PHP Programs

Problem

You want to perform a transaction in a PHP script.

Solution

Use the standard PDO transaction support mechanism.

Discussion

The PDO extension supports a transaction abstraction that can be used to perform transactions. To begin a transaction, use the `beginTransaction()` method. Then, after executing your statements, invoke either `commit()` or `rollback()` to commit or roll

back the transaction. The following code illustrates this. It uses exceptions to detect transaction failure, so it assumes that exceptions are enabled for PDO errors:

```
try
{
  $dbh->beginTransaction ();
  $dbh->exec ("UPDATE money SET amt = amt - 6 WHERE name = 'Eve'");
  $dbh->exec ("UPDATE money SET amt = amt + 6 WHERE name = 'Ida'");
  $dbh->commit ();
}
catch (Exception $e)
{
  print ("Transaction failed, rolling back. Error was:\n");
  print ($e->getMessage () . "\n");
  # empty exception handler in case rollback fails
  try
  {
    $dbh->rollback ();
  }
  catch (Exception $e2) { }
}
```

17.7. Using Transactions in Python Programs

Problem

You want to perform a transaction in a DB API script.

Solution

Use the standard DB API transaction support mechanism.

Discussion

The Python DB API abstraction provides transaction processing control through connection object methods. The DB API specification indicates that database connections should begin with auto-commit mode disabled. Therefore, when you open a connection to the database server, Connector/Python disables auto-commit mode, which implicitly begins a transaction. End each transaction with either `commit()` or `rollback()`. The `commit()` call occurs within a `try` statement, and the `rollback()` occurs within the `except` clause to cancel the transaction if an error occurs:

```
try:
  cursor = conn.cursor()
  # move some money from one person to the other
  cursor.execute("UPDATE money SET amt = amt - 6 WHERE name = 'Eve'")
  cursor.execute("UPDATE money SET amt = amt + 6 WHERE name = 'Ida'")
  cursor.close()
  conn.commit()
```

```
except mysql.connector.Error as e:
  print("Transaction failed, rolling back. Error was:")
  print(e)
  try:  # empty exception handler in case rollback fails
    conn.rollback()
  except:
    pass
```

17.8. Using Transactions in Java Programs

Problem

You want to perform a transaction in a JDBC application.

Solution

Use the standard JDBC transaction support mechanism.

Discussion

To perform transactions in Java, use your `Connection` object to turn off auto-commit mode. Then, after executing your statements, use the object's `commit()` method to commit the transaction or `rollback()` to cancel it. Typically, you execute the statements for the transaction in a `try` block, with `commit()` at the end of the block. To handle failures, invoke `rollback()` in the corresponding exception handler:

```
try
{
  conn.setAutoCommit (false);
  Statement s = conn.createStatement ();
  // move some money from one person to the other
  s.executeUpdate ("UPDATE money SET amt = amt - 6 WHERE name = 'Eve'");
  s.executeUpdate ("UPDATE money SET amt = amt + 6 WHERE name = 'Ida'");
  s.close ();
  conn.commit ();
  conn.setAutoCommit (true);
}
catch (SQLException e)
{
  System.err.println ("Transaction failed, rolling back. Error was:");
  Cookbook.printErrorMessage (e);
  // empty exception handler in case rollback fails
  try
  {
    conn.rollback ();
    conn.setAutoCommit (true);
  }
  catch (Exception e2) { }
}
```

CHAPTER 18

Introduction to MySQL on the Web

18.0. Introduction

This chapter and the next few discuss how MySQL helps you build a better website. One significant benefit is a more interactive site; MySQL makes it easier to provide dynamic content rather than static content. Static content exists as pages in the web server's document tree that are served exactly as is. Visitors can access only the documents that you place in the tree, and changes occur only when you add, modify, or delete those documents. By contrast, dynamic content is created on demand. Rather than opening a file and serving its contents directly to the client, the web server executes a script that generates the page and sends the resulting output. For example, a script can process a keyword request and return a page that lists items in a catalog that match the keyword. Each time a keyword is submitted, the script produces a result appropriate for the request. And that's just for starters; web scripts have access to the power of the programming language in which they're written, so the actions they perform to generate pages can be quite extensive. For example, web scripts are important for form processing, and a single script may be responsible for generating a form and sending it to the user, processing the contents of the form when the user submits it later, and storing the contents in a database. Scripts that operate this way interact with visitors to your website and tailor the information provided according to what they want to see.

This chapter covers the introductory aspects of writing scripts that use MySQL in a web environment. It establishes the groundwork for using your database within the context of web programming. The topics covered here include:

- How web scripting differs from writing static HTML documents or scripts intended to be executed from the command line.
- Prerequisites for running web scripts. In particular, you must have a web server installed and it must be set up to recognize your scripts as programs to be executed, rather than as static files to be served without change over the network.

- How to use each of our API languages to write a short web script that queries the MySQL server and displays the results in a web page.
- How to properly encode output. HTML consists of text to be displayed interspersed with special markup constructs. If the text contains special characters, you must encode them to avoid generating malformed web pages. Each API provides a way to do this.

The following chapters go into more detail on topics such as displaying query results in various formats (paragraphs, lists, tables, and so forth), working with images, form processing, and tracking a user across the course of several page requests as part of a single user session.

This book uses the Apache web server for Perl, Ruby, PHP, and Python scripts. It uses the Tomcat server for Java scripts written using JSP notation. Apache and Tomcat are available from the Apache Software Foundation.

Recipe 18.2 discusses how to configure Apache for Perl, Ruby, PHP, and Python, and how to write a short web script in each language. Recipe 18.3 discusses JSP script writing using Tomcat. Because Apache installations are prevalent, I assume that it's already installed on your system and you just need to configure it. Tomcat is less frequently preinstalled; for additional installation and setup information, read "JSP, JSTL, and Tomcat Primer" on the compnaion website (see the Preface). You can use servers other than Apache and Tomcat, if you adapt the instructions given here.

Scripts for examples in this chapter are located in the `recipes` distribution under the directories named for the servers used to run them. For Perl, Ruby, PHP, and Python examples, look under the *apache* directory. For Java (JSP) examples, look under the *tomcat* directory.

I assume here that you have some basic familiarity with HTML. For Tomcat, it's also helpful to know something about XML because Tomcat's configuration files are written as XML documents, and JSP pages contain elements written using XML syntax. In general, the web scripts in this book produce output that is valid not only as HTML, but as XHTML, the transitional format between HTML and XML. (That's another reason to be familiar with XML.) For example, XHTML requires closing tags, so paragraphs are written with a closing `</p>` tag following the paragraph body. Uses of this output style will be obvious for scripts written using languages like PHP in which the HTML tags are included literally in the script. For interfaces that generate HTML for you, XHTML conformance is a matter of whether the module itself produces XHTML. For example, the Perl CGI.pm module generates XHTML; the Ruby `cgi` module does not.

18.1. Basic Principles of Web Page Generation

Problem

You want to produce a web page from a script, not write a static page manually.

Solution

Write a program that generates the page when it executes. This gives you more control over what is sent to the client than when you write a static page, but may also require that you provide more parts of the response. For example, it may be necessary to write the headers that precede the page body.

Discussion

HTML is a markup language—that's what the "ML" stands for. HTML consists of a mix of plain text to be displayed and special markup indicators or constructs that control how the plain text is displayed. Here is a very simple HTML page that specifies a title in the page header, and a body containing a single paragraph:

```
<html>
<head><title>Web Page Title</title></head>
<body>
<p>Web page body.</p>
</body>
</html>
```

It's possible to write a script that produces that same page, but doing so differs from writing a static page. For one thing, you're writing in two languages at once: the script is written in your programming language, and the script itself writes HTML. Another difference is that you may have to produce more of the response that is sent to the client. When a web server sends a static page to a client, it sends a set of one or more header lines first that provide additional information about the page. For example, an HTML document is preceded by a `Content-Type:` header that lets the client know what kind of information to expect, and a blank line that separates any headers from the page body:

```
Content-Type: text/html

<html>
<head><title>Web Page Title</title></head>
<body>
<p>Web page body.</p>
</body>
</html>
```

To indicate a particular character set encoding, add it to the `Content-Type:` header. For good measure, specify it in a `<meta>` tag as well:

```
Content-Type: text/html; charset=UTF-8

<html>
<head>
…
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
…
</head>
…
```

For static HTML pages, the web server produces header information automatically. When you write a web script, you may need to provide the header information yourself. Some APIs (such as PHP) send a content-type header automatically, but enable you to override the default type. For example, if your script sends a JPEG image to the client instead of an HTML page, the script should change the content type from `text/html` to `image/jpeg`.

Writing web scripts also differs from writing command-line scripts, for both input and output. On the input side, the information given to a web script is provided by the web server rather than by input entered at the command line. This means your scripts do not obtain data using input statements. Instead, the web server puts information into the execution environment of the script, which then extracts that information from its environment and acts on it.

On the output side, command-line scripts typically produce plain-text output. Web scripts can do that, too, or instead produce HTML, images, audio, and so forth. Such output produced in a web environment usually must be highly structured to ensure that it can be understood by the receiving client program.

Any programming language enables output generation using print statements. Some languages also offer special assistance for producing web pages, typically by means of special modules:

- For Perl scripts, a popular module is CGI.pm. It provides features for generating HTML markup, form processing, and more.
- In Ruby, the `cgi` module provides capabilities similar to CGI.pm.
- PHP scripts are written as a mix of HTML and embedded PHP code. That is, you write HTML literally into the script, and then drop into "program mode" whenever you need to generate output by executing code. PHP replaces the code by its output in the resulting page that is sent to the client.
- Python has `cgi` and `urllib` modules that help perform web programming tasks.
- Java scripts written according to the JSP specification can use scripting directives and code embedded within web pages. This is similar to how PHP works.

Before you can run any scripts in a web environment, your web server must be set up properly. Recipes 18.2 and 18.3 provide information about doing this for Apache and Tomcat.

If you run multiple web servers on the same host, they must listen for requests on different port numbers. In a typical configuration, Apache listens on the default HTTP port (80) and Tomcat listens on another port such as 8080. The examples here use a server hostname of *localhost* to represent URLs for scripts processed using Apache and Tomcat. The examples use a different port (8080) for Tomcat scripts. Typical forms for URLs that you'll see in this book are as follows:

```
http://localhost/cgi-bin/my_perl_script.pl
http://localhost/cgi-bin/my_ruby_script.rb
http://localhost/cgi-bin/my_python_script.py
http://localhost/mcb/my_php_script.php
http://localhost:8080/mcb/my_jsp_script.jsp
```

Change the hostname and port number as necessary for pages served by your own web servers.

18.2. Using Apache to Run Web Scripts

Problem

You want to run Perl, Ruby, PHP, or Python programs in a web environment.

Solution

Execute them using the Apache server.

Discussion

This recipe describes how to configure Apache for running Perl, Ruby, PHP, and Python scripts. It also illustrates how to write web-based scripts in each language.

There are typically several directories under the Apache root directory. Here I'll assume that directory to be */usr/local/apache*, although it might differ on your system. For example, on Windows, you might find Apache under *C:\Program Files*. The directories under the Apache root include *bin* (which contains *httpd*—that is, Apache itself—and other Apache-related executable programs), *conf* (for configuration files, notably *httpd.conf*, the primary file used by Apache), *htdocs* (the root of the document tree), and *logs* (for logfiles). The layout might differ on your system. For example, you might find the configuration files in */etc/httpd* and the logs under */var/log/httpd*. Adjust the following instructions accordingly.

To configure Apache for script execution, edit the *httpd.conf* file in the *conf* directory. Typically, executable scripts are identified either by location or by filename suffix. A location can be either language-neutral or language-specific.

Apache configurations often have a *cgi-bin* directory under the Apache root directory in which you can install scripts that should run as external programs. It's configured using a `ScriptAlias` directive:

```
ScriptAlias /cgi-bin/ /usr/local/apache/cgi-bin/
```

The second argument is the actual location of the script directory in your filesystem, and the first is the pathname in URLs that corresponds to that directory. Thus, the directive just shown associates scripts located in */usr/local/apache/cgi-bin* with URLs that have *cgi-bin* following the hostname. If the Ruby script *myscript.rb* is installed in the directory */usr/local/apache/cgi-bin* on the local host, request it with this URL:

```
http://localhost/cgi-bin/myscript.rb
```

When configured this way, the *cgi-bin* directory can contain scripts written in any language. Because of this, the directory is language-neutral, so each script must indicate which language processor executes it. To provide this information, the first line of a script should begin with `#!` followed by the pathname to the appropriate program. For example, a script that begins with the following line is run by Perl:

```
#!/usr/bin/perl
```

Under Unix, you must also make the script executable (use *chmod* +x), or it won't run properly. The `#!` line just shown is appropriate for a system that has Perl installed as */usr/bin/perl*. If your Perl interpreter is installed somewhere else, modify the line accordingly. If you're on a Windows machine with Perl installed as *C:\Perl\bin\perl.exe*, the `#!` line looks like this:

```
#!C:\Perl\bin\perl
```

For Windows, a simpler option is to set up a filename extension association between script names that end with a *.pl* suffix and the Perl interpreter. Then invoking a script with that suffix causes it to be executed by Perl without naming the interpreter.

Directories used only for scripts generally are placed outside of your Apache document tree. As an alternative to using specific directories for scripts, you can identify scripts by filename extension, so that Apache associates files with a particular suffix with a specific language processor. In this case, you can place them anywhere in the document tree. This is the most common way to use PHP. For example, if you have Apache configured with PHP support built in using the *mod_php* module, you can tell it that scripts having names ending with *.php* should be interpreted as PHP scripts. To do so, add this line to *httpd.conf*:

```
AddType application/x-httpd-php .php
```

You may also have to add a `LoadModule` directive for *php*.

With PHP enabled, you can install a PHP script *myscript.php* under *htdocs* (the Apache document root directory). The URL for invoking the script becomes:

```
http://localhost/myscript.php
```

If PHP runs as an external standalone program, you must tell Apache where to find it. For example, if you run Windows and you have PHP installed as *C:\Php\php.exe*, put the following lines in *httpd.conf* (note the use of forward slashes in the pathnames rather than backslashes):

```
ScriptAlias /php/ "C:/Php/"
AddType application/x-httpd-php .php
Action application/x-httpd-php /php/php.exe
```

For purposes of showing URLs in examples, I'll assume that Perl, Ruby, and Python scripts are in your *cgi-bin* directory, and that PHP scripts are in the *mcb* directory of your document tree, identified by the *.php* extension. That means the URLs for scripts in these languages look like this:

```
http://localhost/cgi-bin/myscript.pl
http://localhost/cgi-bin/myscript.rb
http://localhost/cgi-bin/myscript.py
http://localhost/mcb/myscript.php
```

Adjust the pathnames as necessary for your own system.

If you plan to use a similar setup, make sure to have a *cgi-bin* directory that Apache knows about, and an *mcb* directory under your Apache document root. Then, to deploy Perl, Ruby, or Python web scripts, install them in the *cgi-bin* directory. To deploy PHP scripts, install them in the *mcb* directory.

Some of the scripts use modules or library files that are specific to this book. If you have these files installed in a library directory that your language processors search by default, they should be found automatically. Otherwise, you must indicate where the files are located. An easy way to do this is by using `SetEnv` directives in your *httpd.conf* file to set environment variables that can be seen by your scripts when Apache invokes them. (To use the `SetEnv` directive, the `mod_env` Apache module must be enabled.) For example, if you install library files in */usr/local/lib/mcb*, the following directives enable Perl, Ruby, and Python scripts to find them:

```
SetEnv PERL5LIB /usr/local/lib/mcb
SetEnv RUBYLIB /usr/local/lib/mcb
SetEnv PYTHONPATH /usr/local/lib/mcb
```

For PHP, add */usr/local/lib/mcb* to the value of `include_path` in your *php.ini* configuration file.

For background information on library-related environment variables and the *php.ini* file, see Recipe 2.3.

After configuring Apache to support script execution, restart it. Then you can begin to write scripts that generate web pages. The remainder of this section describes how to do so for Perl, Ruby, PHP, and Python. The example for each language connects to the MySQL server, runs a simple query, and displays the results in a web page. The scripts shown here indicate whether any additional modules or libraries are typically required. (Later sections generally assume that the proper modules have been included and show only script fragments.)

Before we proceed further, here are some debugging tips:

- If you request a web script and get an error page in response, the Apache error log is a useful source of diagnostic information. A common name for this log is *error_log* in the *logs* directory. If you don't find any such file, check *httpd.conf* for an `ErrorLog` directive to see where Apache logs errors.
- Sometimes it's helpful to directly examine the output that a web script generates. To do this, invoke the script from the command line. You'll see the HTML produced by the script, as well as any error messages that it might print. Some web modules expect to see a parameter string, and might even prompt you for one when you invoke the script at the command line. When this is the case, you might be able to supply the parameters as an argument on the command line to avoid the prompt. For example, the Ruby `cgi` module expects to see parameters, and prompts you if they are missing:

  ```
  % myscript.rb
  (offline mode: enter name=value pairs on standard input)
  ```

  At the prompt, enter the parameter values and then enter Ctrl-D (EOF). To avoid the prompt, supply the parameters on the command line:

  ```
  % myscript.rb "param1=val1;param2=val2;param3=val3"
  ```

  To specify "no parameters" explicitly, provide an empty argument:

  ```
  % myscript.rb ""
  ```

Web Security Note

Under Unix, scripts are associated with particular user and group IDs when they execute. Scripts that you execute from the command line run with your own user and group IDs, and have the filesystem privileges associated with your account. Scripts executed by a web server don't run with your user and group ID, nor will they have your privileges. Instead, they run under the user and group ID of the account the web server has been set to run as, and with that account's privileges. (To determine what account this is, look

for `User` and `Group` directives in the *httpd.conf* configuration file.) If you expect web scripts to read and write files, those files must be accessible to the account used to run the web server. For example, if your server runs under the `nobody` account and you want a script to be able to store uploaded image files into a directory called *uploads* in the document tree, that directory must be readable and writable by the `nobody` user.

Another implication is that if other people can write scripts to be executed by your web server, those scripts too run as `nobody` and they can read and write the same files as your own scripts. That is, files used by your scripts cannot be considered private only to your scripts. A solution to this problem is to use the Apache suEXEC mechanism. (If you use an ISP for web hosting, suEXEC might be enabled already.)

Perl

Our first web-based Perl script retrieves and displays a list of tables in the `cookbook` database. It produces HTML elements using the CGI.pm module, which makes it easy to write web scripts without writing literal HTML tags. CGI.pm provides an object-oriented interface and a function call interface, so you can use it to write web pages in either of two styles. Here's a script, *show_tables_oo.pl*, that produces the table listing using the object-oriented interface:

```
#!/usr/bin/perl
# show_tables_oo.pl: Display names of tables in cookbook database
# (uses the CGI.pm object-oriented interface)

use strict;
use warnings;
use CGI;
use Cookbook;

# Create CGI object for accessing CGI.pm methods

my $cgi = new CGI;

# Print header, blank line, and initial part of page

print $cgi->header ();
print $cgi->start_html (-title => "Tables in cookbook Database");

print $cgi->p ("Tables in cookbook database:");

# Connect to database, display table list, disconnect

my $dbh = Cookbook::connect ();
my $stmt = "SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
            WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
while (my @row = $sth->fetchrow_array ())
```

```
{
  print $row[0], $cgi->br ();
}
$dbh->disconnect ();

# Print page trailer

print $cgi->end_html ();
```

To try the script, install it in your *cgi-bin* directory and request it from your browser as follows:

```
http://localhost/cgi-bin/show_tables_oo.pl
```

The script includes the CGI.pm module with a `use CGI` statement, and then creates a `CGI` object, `$cgi`, through which it invokes the various HTML-generation calls. `header()` generates the `Content-Type:` header and `start_html()` produces the initial page tags up through the opening `<body>` tag. After generating the first part of the page, *show_tables_oo.pl* retrieves and displays information from the server. Each table name is followed by a `<br />` tag, produced by invoking the `br()` method. `end_html()` produces the closing `</body>` and `</html>` tags.

CGI.pm calls often take multiple parameters, many of which are optional. To enable you to specify just those parameters you need, CGI.pm understands *`-name => value`* notation in parameter lists. For example, in the `start_html()` call, the `title` parameter sets the page title. The *`-name => value`* notation also permits parameters to be specified in any order.

To use the CGI.pm function call interface rather than the object-oriented interface, write scripts a little differently. The `use` line that references CGI.pm should import the method names into your script's namespace so that you can invoke them directly as functions without having to create a `CGI` object. For example, to import the most commonly used methods, the script should include this statement:

```
use CGI qw(:standard);
```

The following script, *show_tables_fc.pl*, is the function call equivalent of the *show_tables_oo.pl* script just shown. It uses the same CGI.pm calls, but invokes them as standalone functions rather than through a `$cgi` object:

```
#!/usr/bin/perl
# show_tables_fc.pl: Display names of tables in cookbook database
# (use the CGI.pm function-call interface)

use strict;
use warnings;
use CGI qw(:standard); # import standard method names into script namespace
use Cookbook;

# Print header, blank line, and initial part of page
```

```
print header ();
print start_html (-title => "Tables in cookbook Database");

print p ("Tables in cookbook database:");

# Connect to database, display table list, disconnect

my $dbh = Cookbook::connect ();
my $stmt = "SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
            WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
while (my @row = $sth->fetchrow_array ())
{
  print $row[0], br ();
}
$dbh->disconnect ();

# Print page trailer

print end_html ();
```

Install the *show_tables_fc.pl* script in your *cgi-bin* directory and request it from your browser to verify that it produces the same output as *show_tables_oo.pl*.

This book uses the CGI.pm function call interface for Perl-based web scripts from this point on. You can get more information about CGI.pm at the command line by using the following commands to read the installed documentation:

```
% perldoc CGI
% perldoc CGI::Carp
```

Documentation is also available online from CPAN.

Ruby

The Ruby `cgi` module provides an interface to HTML-generating methods. To use it, create a `CGI` object and invoke its methods to produce HTML page elements. Method names correspond to the HTML elements they produce. Their invocation syntax follows these principles:

- If an element should have attributes, pass them as arguments to the method.
- If the element has body content, specify the content in a code block associated with the method call.

For example, the following method call produces a `<P>` element that includes an `align` attribute and content of "This is a sentence":

```
cgi.p("align" => "left") { "This is a sentence." }
```

The output looks like this:

```
<P align="left">This is a sentence.</P>
```

To display generated HTML content, pass it in a code block to the `cgi.out` method. The following Ruby script, *show_tables.rb*, retrieves a list of tables in the `cookbook` database and displays them as an HTML document:

```
#!/usr/bin/ruby -w
# show_tables.rb: Display names of tables in cookbook database

require "cgi"
require "Cookbook"

# Connect to database, generate table list, disconnect

dbh = Cookbook.connect
stmt = "SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME"
rows = dbh.select_all(stmt)
dbh.disconnect

cgi = CGI.new("html4")

cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { "Tables in cookbook Database" }
    } +
    cgi.body() {
      cgi.p { "Tables in cookbook Database:" } +
      rows.collect { |row| row[0] + cgi.br }.join
    }
  }
}
```

The `collect` method iterates through the row array containing the table names and produces a new array containing each name with a `<br>` appended to it. The `join` method concatenates the strings in the resulting array.

The script includes no explicit code for producing the `Content-Type:` header because `cgi.out` generates one.

Install the script in your *cgi-bin* directory and request it from your browser as follows:

```
http://localhost/cgi-bin/show_tables.rb
```

If you invoke Ruby web scripts from the command line so that you can examine the generated HTML, you'll see that the HTML is all on one line and is difficult to read. To make the output easier to understand, process it through the `CGI.pretty` utility method, which adds line breaks and indentation. Suppose that your page output call looks like this:

```
cgi.out {
  page content here
}
```

To change the call to use `CGI.pretty`, write it like this:

```
cgi.out {
  CGI.pretty(page content here)
}
```

PHP

PHP doesn't provide much in the way of tag shortcuts, which is surprising given that language's web orientation. On the other hand, because PHP is an embedded language, you can simply write your HTML literally in your script without using `print` statements. Here's a *show_tables.php* script that shifts back and forth between HTML mode and PHP mode:

```
<?php
# show_tables.php: Display names of tables in cookbook database

require_once "Cookbook.php";
?>

<html>
<head><title>Tables in cookbook Database</title></head>
<body>

<p>Tables in cookbook database:</p>

<?php
# Connect to database, display table list, disconnect
$dbh = Cookbook::connect ();
$stmt = "SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
         WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME";
$sth = $dbh->query ($stmt);
while (list ($tbl_name) = $sth->fetch (PDO::FETCH_NUM))
  print ($tbl_name . "<br />");
$dbh = NULL;
?>

</body>
</html>
```

To try the script, put it in the *mcb* directory of your web server's document tree and request it from your browser as follows:

```
http://localhost/mcb/show_tables.php
```

The PHP script includes no code to produce the `Content-Type:` header because PHP produces one automatically. (To override this behavior and produce your own headers, consult the `header()` function section in the PHP manual.)

Except for the break tags, *show_tables.php* includes HTML content by writing it outside of the `<?php` and `?>` tags so that the PHP interpreter simply writes it without interpretation. Here's a different version that produces all the HTML using `print` statements:

```
<?php
# show_tables_print.php: Display names of tables in cookbook database
# using print() to generate all HTML

require_once "Cookbook.php";

print ("<html>");
print ("<head><title>Tables in cookbook Database</title></head>");
print ("<body>");
print ("<p>Tables in cookbook database:</p>");

# Connect to database, display table list, disconnect
$dbh = Cookbook::connect ();
$stmt = "SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
         WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME";
$sth = $dbh->query ($stmt);
while (list ($tbl_name) = $sth->fetch (PDO::FETCH_NUM))
  print ($tbl_name . "<br />");
$dbh = NULL;

print ("</body>");
print ("</html>");
?>
```

Sometimes it makes sense to use one approach, sometimes the other—and sometimes both within the same script. If a section of HTML contains no references to variable or expression values, it can be clearer to write it in HTML mode. Otherwise it may be clearer to write it using `print` or `echo` statements, to avoid switching between HTML and PHP modes frequently.

Python

A standard installation of Python includes `cgi` and `urllib` modules that are useful for web programming. However, we don't actually need them yet because the only web-related activity of our first Python web script is to generate some simple HTML. Here's a Python version of the MySQL table-display script:

```
#!/usr/bin/python
# show_tables.py: Display names of tables in cookbook database

import cookbook

# Print header, blank line, and initial part of page

print('''Content-Type: text/html

<html>
```

```
<head><title>Tables in cookbook Database</title></head>
<body>

<p>Tables in cookbook database:</p>
''')

# Connect to database, display table list, disconnect

conn = cookbook.connect()
cursor = conn.cursor()
stmt = '''
  SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME
'''
cursor.execute(stmt)
for (tbl_name, ) in cursor:
  print("%s<br />" % tbl_name)
cursor.close()
conn.close()

# Print page trailer
print('''
</body>
</html>
''')
```

Put the script in Apache's *cgi-bin* directory and request it from your browser like this:

```
http://localhost/cgi-bin/show_tables.py
```

18.3. Using Tomcat to Run Web Scripts

Problem

You want to run Java-based programs in a web environment.

Solution

Write programs using JSP notation and execute them using a servlet container such as Tomcat.

Discussion

As described in Recipe 18.2, Apache can be used to run Perl, Ruby, PHP, and Python scripts. Java requires a different approach because Apache doesn't serve JSP pages. Instead, we'll use Tomcat, a server designed for processing Java in a web environment. Apache and Tomcat are very different servers, but there is a familial relationship—like Apache, Tomcat is a development effort of the Apache Software Foundation.

Java JSP scripts are compiled into Java servlets and run inside a process known as a servlet container. The first time a client requests a given JSP page, the container compiles the page into a servlet in the form of executable Java byte code before loading and running it. The container caches the byte code so that the script can run directly with no compilation phase for subsequent requests. If you modify the script, the container notices this when the next request arrives, recompiles the script into a new servlet, and reloads it. The JSP approach provides a significant advantage over writing servlets directly, because you need not compile code yourself or handle servlet loading and unloading. Tomcat handles the responsibilities of both the servlet container and the web server that communicates with the container.

This section provides an overview of JSP programming with Tomcat, but makes several assumptions:

- You are familiar with the concepts underlying JavaServer Pages, such as what a servlet container is, what an application context is, and what the basic JSP scripting elements are.
- The Tomcat server has been installed so that you can execute JSP pages, and you know how to start and stop it.
- You are familiar with the Tomcat *webapps* directory and how Tomcat applications are structured.
- You know what a tag library is, how to use one, and are familiar with the JSP Standard Tag Library (JSTL).

I recognize that is a lot to assume. If you're unfamiliar with JSP or JSTL, or need instructions for installing Tomcat, read "JSP, JSTL, and Tomcat Primer" on the companion website for the necessary background information (see the Preface).

Once you have Tomcat in place, install the following components so that you can work through the JSP examples in this book:

- The `mcb` sample application. This is located in the *tomcat* directory of the `recipes` distribution.
- A MySQL JDBC driver. You might already have one installed for use with the scripts in earlier chapters, but Tomcat needs a copy, too. This book uses MySQL Connector/J.
- The JSTL tag library, which contains tags for performing database activities, conditional testing, and iterative operations within JSP pages.

This section discusses how to install these components and describes how to write the JSP equivalent of the MySQL table-display script that was implemented in Recipe 18.2 using Perl, Ruby, PHP, and Python.

Installing the mcb application

Web applications for Tomcat typically are packaged as WAR (web archive) files and installed under its *webapps* directory, which is roughly analogous to Apache's *htdocs* document root directory. The `recipes` distribution includes a sample application named `mcb` to use for the JSP examples described here. Look in the distribution's *tomcat* directory, where you will find a file named *mcb.war*. Copy that file to Tomcat's *webapps* directory and restart Tomcat.

As distributed, Tomcat is configured by default to look for WAR files under *webapps* when it starts and automatically unpack any that have not already been unpacked. This means that restarting Tomcat after copying *mcb.war* to the *webapps* directory should be enough to unpack the `mcb` application. When Tomcat finishes its startup sequence, look under *webapps* and you should see a new *mcb* directory under which are all the files contained in *mcb.war*. Explore the *mcb* directory if you like. It should contain several files that clients can request using a browser. There should also be a *WEB-INF* subdirectory, which is used for information that is private—that is, available for use by scripts in the *mcb* directory, but not directly accessible by clients.

Next, to verify that Tomcat can serve pages from the `mcb` application context, request some of them from your browser. The main `mcb` page is:

```
http://localhost:8080/mcb/
```

The following URLs request in turn a simple static HTML page, servlet, and JSP page (each is available from the main page):

```
http://localhost:8080/mcb/simple.html
http://localhost:8080/mcb/servlet/SimpleServlet
http://localhost:8080/mcb/simple.jsp
```

Adjust the hostname and port number in the URLs appropriately for your installation.

Installing the JDBC driver

Some JSP pages in the `mcb` application need a JDBC driver for connecting to the `cookbook` database. This book uses the MySQL Connector/J driver.

To install Connector/J for use by Tomcat applications, place a copy of it in Tomcat's directory tree. Assuming that the driver is packaged as a JAR file (as is the case for Connector/J), there are different places under the Tomcat root directory where you can install it, depending on how visible you want the driver to be:

- To make the driver available only to the `mcb` application, place it in the *mcb/WEB-INF/lib* directory under Tomcat's *webapps* directory.
- To make the driver available both to Tomcat and to applications, place it in the *lib* directory under the Tomcat root.

I recommend installing the driver in the *lib* directory. That gives it the most global visibility (accessible by Tomcat and by applications), and you need install it only once. If you enable the driver only for the `mcb` application by placing a copy in *mcb/WEB-INF/lib*, but then develop other applications that use MySQL, you must either copy the driver into those applications or move it to a more global location.

Making the driver more globally accessible also is useful if you think that at some point you may elect to use JDBC-based session management (seeRecipe 21.4) or realm authentication. Those activities are handled by Tomcat itself above the application level, so Tomcat needs access to the driver to carry them out.

After installing Connector/J, restart Tomcat. Then request the following `mcb` application page to verify that Tomcat can find and use Connector/J:

```
http://localhost:8080/mcb/jdbc_test.jsp
```

You might need to modify *jdbc_test.jsp* to change the connection parameters.

Installing the JSTL distribution

Many of the scripts that are part of the `mcb` sample application use JSTL, a popular tag library. It's necessary to install JSTL or those scripts won't work. To install a tag library into an application context, copy the library's JAR file or files into the application's *WEB-INF/lib* directory. The following instructions describe how to install JSTL for use with the `mcb` application:

1. Make sure that the *mcb.war* file has been unpacked to create the `mcb` application directory hierarchy under the Tomcat *webapps* directory. (Refer back to "Installing the mcb application.") The JSTL files must be installed under the *mcb* directory, which does not exist until *mcb.war* has been unpacked.
2. Get JSTL from the Apache Standard Taglibs project page, which has a download link from which you can get a binary JSTL distribution. Get version 1.1.2 or higher.
3. Unpack the JSTL distribution into some convenient location, preferably outside of the Tomcat hierarchy. If you use a ZIP archive, you can unpack it with the *jar* utility or any other program that understands ZIP format (such as the Windows *WinZip* application). For example, with *jar*, use the following command, adjusting the filename as necessary:

   ```
   % jar xf jakarta-taglibs-standard.zip
   ```

4. Unpacking the distribution creates a directory containing several directories and files. Change location into the *lib* directory and copy the *jstl.jar* and *standard.jar* JAR files to the *mcb/WEB-INF/lib* directory. Those files contain the classes that implement the JSTL tag actions, and tag library descriptor files that define the interface for the actions implemented by the classes.

5. The *mcb/WEB-INF* directory contains a file named *jstl-mcb-setup.inc*. This file is not part of JSTL itself, but it contains a JSTL `<sql:setDataSource>` tag used by many of the `mcb` JSP pages to set up a data source for connecting to the `cookbook` database. The file looks like this:

```
<sql:setDataSource
  var="conn"
  driver="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost/cookbook"
  user="cbuser"
  password="cbpass"
/>
```

   Edit the `url`, `user`, and `password` tag attributes if necessary to change the connection parameters to those that you use for accessing the `cookbook` database. Do not change the `var` attribute, which names the variable to associate with the connection. By convention, `mcb` JSP pages in *MySQL Cookbook* use the variable `conn`; tags occurring later in the page that require a data source refer to the connection using the expression `${conn}`.

6. The JSTL distribution also includes WAR files containing documentation and examples (*standard-doc.war* and *standard-examples.war*). If you want to install these, copy them into Tomcat's *webapps* directory. (I recommand that you install the documentation so that you can access it locally from your own server. It's useful to install the examples as well because they provide helpful demonstrations showing how to use JSTL tags in JSP pages.)
7. Restart Tomcat so it notices the changes you just made to the `mcb` application and unpacks the WAR files containing the JSTL documentation and examples.

After installing JSTL and restarting Tomcat, request the following `mcb` application page to verify that Tomcat can find and use the JSTL tag library properly:

```
http://localhost:8080/mcb/jdbc_jstl_test.jsp
```

Use these URLs to access the documentation and examples:

```
http://localhost:8080/standard-doc/
http://localhost:8080/standard-examples/
```

Writing a MySQL script using JSP and JSTL

Recipe 18.2 shows how to write Perl, Ruby, PHP, and Python versions of a script to display the names of the tables in the `cookbook` database. With the JSTL tags, we can write a corresponding JSP page that provides that information:

```
<%-- show_tables.jsp: Display names of tables in cookbook database --%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ include file="/WEB-INF/jstl-mcb-setup.inc" %>
```

```
<html>
<head><title>Tables in cookbook Database</title></head>
<body>

<p>Tables in cookbook database:</p>

<sql:query dataSource="${conn}" var="rs">
  SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME
</sql:query>

<c:forEach items="${rs.rowsByIndex}" var="row">
  <c:out value="${row[0]}"/><br />
</c:forEach>

</body>
</html>
```

The `taglib` directives identify which tag libraries the page uses, and the `include` directive pulls in the code that sets up a data source for accessing the `cookbook` database. The rest of the script generates the page content.

Assuming that you've installed the *mcb.war* file in your Tomcat server's *webapps* directory as described earlier, you should find the *show_tables.jsp* script in the *mcb* subdirectory. Request it from your browser as follows:

```
http://localhost:8080/mcb/show_tables.jsp
```

The JSP script does not produce any `Content-Type:` header explicitly. The JSP engine produces a default header with a content type of `text/html` automatically.

18.4. Encoding Special Characters in Web Output

Problem

Certain characters are special in web pages and must be encoded if you want to display them literally. Because database content often contains instances of these characters, scripts that include query results in web pages should encode those results to prevent browsers from misinterpreting the information.

Solution

Use the methods provided by your API for performing HTML-encoding and URL-encoding.

Discussion

HTML is a markup language: it uses certain characters as markers that have a special meaning. To include literal instances of these characters in a page, you must encode them so that they are not interpreted as having their special meanings. For example, encode < as `<` to keep a browser from interpreting it as the beginning of a tag. Furthermore, there are actually two kinds of encoding, depending on the context in which you use a character. One encoding is appropriate for HTML text, another is used for text that is part of a URL in a hyperlink.

The MySQL table-display scripts shown in Recipes 18.2 and 18.3 are simple demonstrations of how to produce web pages using programs. But with one exception, the scripts have a common failing: they take no care to properly encode special characters that occur in the information retrieved from the MySQL server. (The exception is the JSP version of the script. The `<c:out>` tag used there handles encoding automatically, as we'll discuss shortly.)

As it happens, I deliberately chose information to display that is unlikely to contain any special characters, so the scripts should work properly even in the absence of any encoding. However, in the general case, it's unsafe to assume that a query result contains no special characters, so you must be prepared to encode it for display in a web page. Neglecting to do this may result in scripts generating pages containing malformed HTML that displays incorrectly.

This recipe describes how to handle special characters, beginning with some general principles, then discusses how each API implements encoding support. The API-specific examples show how to process information drawn from a database table, but they can be adapted to any content you include in a web page, no matter its source.

General encoding principles

One form of encoding applies to characters used in writing HTML constructs; another applies to text included in URLs. It's important to understand this distinction to avoid encoding text the wrong way.

Encoding text for inclusion in a web page is an entirely different issue from encoding special characters in data values for inclusion in an SQL statement. Recipe 2.5 discusses the latter technique.

Encoding characters that are special in HTML. HTML markup uses < and > characters to begin and end tags, & to begin special entity names (such as ` ` to signify a nonbreaking space), and " to quote attribute values in tags (such as `<p align="left">`). Consequently, to display literal instances of these characters, you should encode them

as HTML entities so that browsers or other clients understand your intent. To do this, convert the special characters <, >, &, and " to the corresponding HTML entity designators shown in the following table.

Special character	HTML entity
<	<
>	>
&	&
"	"

Suppose that you want to display the following string literally in a web page:

```
Paragraphs begin and end with <p> & </p> tags.
```

If you send this text to the client browser exactly as shown, the browser will misinterpret it: the and tags will be taken as paragraph markers and the & may be taken as the beginning of an HTML entity designator. To display the string the way you intend, encode the special characters as the `<`, `>`, and `&` entities:

```
Paragraphs begin and end with &lt;p&gt; &amp; &lt;/p&gt; tags.
```

The principle of encoding text this way is also useful within tags. For example, HTML tag attribute values usually are enclosed within double quotes, so it's important to perform HTML-encoding of attribute values. Suppose that you want to include a text input box in a form, and you want to provide an initial value of `Rich "Goose" Gossage` to be displayed in the box. You cannot write that value literally in the tag like this:

```
<input type="text" name="player_name" value="Rich "Goose" Gossage" />
```

The problem here is that the double-quoted `value` attribute includes internal double quotes, which makes the `<input>` tag malformed. To write it properly, encode the double quotes:

```
<input type="text" name="player_name" value="Rich &quot;Goose&quot; Gossage" />
```

When a browser receives this text, it decodes the `&quot;` entities back to " characters and interprets the `value` attribute value correctly.

Encoding characters that are special in URLs. URLs for hyperlinks that occur within HTML pages have their own syntax and their own encoding. This encoding applies to attributes within several tags:

```
<a href="URL">
<img src="URL">
<form action="URL">
<frame src="URL">
```

Many characters have special meaning within URLs, such as `:`, `/`, `?`, `=`, `&`, and `;`. The following URL contains some of these characters:

```
http://localhost/myscript.php?id=428&name=Gandalf
```

Here the `:` and `/` characters segment the URL into components, the `?` character indicates that parameters are present, and the `&` character separates the parameters, each specified as a *`name=value`* pair. (The `;` character is not present in the URL just shown, but commonly is used instead of `&` to separate parameters.) To include any of these characters literally within a URL, you must encode them to prevent the browser from interpreting them with their usual special meaning. Other characters such as spaces require special treatment as well. Spaces are not permitted within a URL, so if you want to reference a page named *my home page.html* on the local host, the URL in the following hyperlink won't work:

```
<a href="http://localhost/my home page.html">My Home Page</a>
```

URL-encoding for special and reserved characters converts each such character to `%` followed by two hexadecimal digits representing the character's ASCII code. For example, the ASCII value of the space character is 32 decimal, or 20 hexadecimal, so write the preceding hyperlink like this:

```
<a href="http://localhost/my%20home%20page.html">My Home Page</a>
```

Sometimes you'll see spaces encoded as `+` in URLs. That is legal, too.

Use the appropriate encoding method for the context:. Be sure to encode information properly for the context in which you use it. Suppose that you want to create a hyperlink to trigger a search for items matching a search term, and you want the term itself to appear as the link label that is displayed in the page. In this case, the term appears as a parameter in the URL, and also as HTML text between the `<a>` and `</a>` tags. If the search term is "cats & dogs", the unencoded hyperlink construct looks like this:

```
<a href="/cgi-bin/myscript?term=cats & dogs">cats & dogs</a>
```

That is incorrect because `&` is special in both contexts and the spaces are special in the URL. Write the link like this instead:

```
<a href="/cgi-bin/myscript?term=cats%20%26%20dogs">cats &amp; dogs</a>
```

Here, `&` is HTML-encoded as `&amp;` for the link label, and is URL-encoded as `%26` for the URL, which also includes spaces encoded as `%20`.

Granted, it's a pain to encode text before writing it to a web page, and sometimes you know enough about a value that you can skip the encoding (see the following sidebar). Otherwise, encoding is the safe thing to do. Fortunately, most APIs provide functions to do the work for you. This means you need not know every character that is special in a given context. You just need to know which kind of encoding to perform, so that you can call the appropriate function to produce the intended result.

Must You Always Encode Web Page Output?

If you *know* a value is legal in a particular context within a web page, you need not encode it. For example, if you obtain a value from an integer-valued column in a database table that cannot be `NULL`, it must necessarily be an integer. No HTML- or URL-encoding is needed to include the value in a web page, because digits are not special in HTML text or within URLs. On the other hand, suppose that you solicit an integer value using a field in a web form. You might be expecting the user to provide an integer, but the user might be confused and enter an illegal value. You could handle this by displaying an error page that shows the value and explains that it's not an integer. But if the value contains special characters and you don't encode it, the page won't display the value properly, possibly further confusing the user.

Encoding special characters using web APIs

The following encoding examples show how to retrieve values from MySQL and perform both HTML-encoding and URL-encoding on them to generate hyperlinks. Each example reads a table named `phrase` that contains short phrases and then uses its contents to construct hyperlinks that point to a (hypothetical) script that searches for instances of the phrases in some other table. The table contains the following rows:

```
mysql> SELECT phrase_val FROM phrase ORDER BY phrase_val;
+----------------------+
| phrase_val           |
+----------------------+
| are we "there" yet?  |
| cats & dogs          |
| rhinoceros           |
| whole > sum of parts |
+----------------------+
```

The goal here is to generate a list of hyperlinks using each phrase both as the hyperlink label (which requires HTML-encoding) and in the URL as a parameter to the search script (which requires URL-encoding). The resulting links look something like this:

```
<a href="/cgi-bin/mysearch.pl?phrase=are%20we%20%22there%22%20yet%3F">
are we &quot;there&quot; yet?</a>
<a href="/cgi-bin/mysearch.pl?phrase=cats%20%26%20dogs">
cats &amp; dogs</a>
<a href="/cgi-bin/mysearch.pl?phrase=rhinoceros">
rhinoceros</a>
<a href="/cgi-bin/mysearch.pl?phrase=the%20whole%20%3E%20sum%20of%20parts">
whole &gt; sum of parts</a>
```

The initial part of the `href` attribute value will vary per API. Also, the links produced by some APIs will look slightly different because they encode spaces as `+` rather than as `%20`.

Perl. The Perl CGI.pm module provides two methods, `escapeHTML()` and `escape()`, that handle HTML-encoding and URL-encoding. There are three ways to use these methods to encode a string `$str`:

- Invoke `escapeHTML()` and `escape()` as CGI class methods using a `CGI::` prefix:

```
use CGI;
printf "%s\n%s\n", CGI::escape ($str), CGI::escapeHTML ($str);
```

- Create a `CGI` object and invoke `escapeHTML()` and `escape()` as object methods:

```
use CGI;
my $cgi = new CGI;
printf "%s\n%s\n", $cgi->escape ($str), $cgi->escapeHTML ($str);
```

- Import the names explicitly into your script's namespace. In this case, neither a `CGI` object nor the `CGI::` prefix is necessary and you invoke the methods as standalone functions. The following example imports the two method names in addition to the set of standard names:

```
use CGI qw(:standard escape escapeHTML);
printf "%s\n%s\n", escape ($str), escapeHTML ($str);
```

I prefer the last alternative because it is consistent with the CGI.pm function call interface that you use for other imported method names. Just remember to include the encoding method names in the `use CGI` statement for any Perl script that requires them, or you'll get "undefined subroutine" errors when the script executes.

The following code reads the rows of the `phrase` table and produces hyperlinks from them using `escapeHTML()` and `escape()`:

```
my $stmt = "SELECT phrase_val FROM phrase ORDER BY phrase_val";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
while (my ($phrase) = $sth->fetchrow_array ())
{
  # URL-encode the phrase value for use in the URL
  my $url = "/cgi-bin/mysearch.pl?phrase=" . escape ($phrase);
  # HTML-encode the phrase value for use in the link label
  my $label = escapeHTML ($phrase);
  print a ({-href => $url}, $label), br ();
}
```

Ruby. The Ruby `cgi` module contains two methods, `CGI.escapeHTML()` and `CGI.escape()`, that perform HTML-encoding and URL-encoding. However, both methods raise an exception unless the argument is a string. To deal with this, apply the `to_s` method to any argument that might not be a string, to force it to string form and convert `nil` to the empty string. For example:

```
stmt = "SELECT phrase_val FROM phrase ORDER BY phrase_val"
dbh.execute(stmt) do |sth|
  sth.fetch do |row|
    # make sure that the value is a string
    phrase = row[0].to_s
    # URL-encode the phrase value for use in the URL
    url = "/cgi-bin/mysearch.rb?phrase=" + CGI.escape(phrase)
    # HTML-encode the phrase value for use in the link label
    label = CGI.escapeHTML(phrase)
    page << cgi.a("href" => url) { label } + cgi.br
  end
end
```

`page` is used here as a variable that "accumulates" page content and that eventually you pass to `cgi.out` to display the page.

PHP. In PHP, the `htmlspecialchars()` and `urlencode()` functions perform HTML-encoding and URL-encoding. Use them as follows:

```
$stmt = "SELECT phrase_val FROM phrase ORDER BY phrase_val";
$sth = $dbh->query ($stmt);
while (list ($phrase) = $sth->fetch (PDO::FETCH_NUM))
{
  # URL-encode the phrase value for use in the URL
  $url = "/mcb/mysearch.php?phrase=" . urlencode ($phrase);
  # HTML-encode the phrase value for use in the link label
  $label = htmlspecialchars ($phrase);
  printf ('<a href="%s">%s</a><br />', $url, $label);
}
```

Python. In Python, the `cgi` and `urllib` modules contain the relevant encoding methods. `cgi.escape()` and `urllib.quote()` perform HTML-encoding and URL-encoding. However, both methods raise an exception unless the argument is a string. To deal with this, apply the `str()` method to any argument that might not be a string, to force it to string form and convert `None` to the string `"None"`. (If you want `None` to convert to the empty string, you must test for it explicitly.) For example:

```
import cgi
import urllib

stmt = "SELECT phrase_val FROM phrase ORDER BY phrase_val"
cursor = conn.cursor()
cursor.execute(stmt)
for (phrase,) in cursor:
  # make sure that the value is a string
  phrase = str(phrase)
  # URL-encode the phrase value for use in the URL
  url = "/cgi-bin/mysearch.py?phrase=" + urllib.quote(phrase)
  # HTML-encode the phrase value for use in the link label
  label = cgi.escape(phrase, 1)
```

```
    print('<a href="%s">%s</a><br />' % (url, label))
cursor.close()
```

The first argument to `cgi.escape()` is the string to be HTML-encoded. By default, this function converts <, >, and & characters to their corresponding HTML entities. To tell `cgi.escape()` to also convert double quotes to the `&quot;` entity, pass a second argument of `1`, as shown in the example. This is especially important if you're encoding values to be placed within a double-quoted tag attribute.

Java. The `<c:out>` JSTL tag automatically performs HTML-encoding for JSP pages. (Strictly speaking, it performs XML-encoding, but the set of characters affected is <, >, &, ", and ', which includes all those needed for HTML-encoding.) By using `<c:out>` to display text in a web page, you need not think about converting special characters to HTML entities. If for some reason you want to suppress encoding, invoke `<c:out>` with an `encodeXML` attribute value of `false`:

```
<c:out value="value to display" encodeXML="false"/>
```

To URL-encode parameters for inclusion in a URL, use the `<c:url>` tag. Specify the URL string in the tag's `value` attribute, and include any parameter values and names in `<c:param>` tags in the body of the `<c:url>` tag. A parameter value can be given either in the `value` attribute of a `<c:param>` tag or in its body. Here's an example that shows both uses:

```
<c:url var="urlStr" value="myscript.jsp">
  <c:param name="id" value ="47"/>
  <c:param name="color">sky blue</c:param>
</c:url>
```

This URL-encodes the values of the `id` and `color` parameters and adds them to the end of the URL. The result is placed in an object named `urlStr`, which you can display as follows:

```
<c:out value="${urlStr}"/>
```

The `<c:url>` tag does not encode special characters such as spaces in the string supplied in its `value` attribute. You must encode them yourself, so it's probably best to avoid creating pages with spaces in their names.

To display entries from the `phrase` table, use the `<c:out>` and `<c:url>` tags as follows:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT phrase_val FROM phrase ORDER BY phrase_val
</sql:query>

<c:forEach items="${rs.rows}" var="row">
  <%-- URL-encode the phrase value for use in the URL --%>
```

```
    <c:url var="urlStr" value="/mcb/mysearch.jsp">
      <c:param name="phrase" value ="${row.phrase_val}"/>
    </c:url>
    <a href="<c:out value="${urlStr}"/>">
      <%-- HTML-encode the phrase value for use in the link label --%>
      <c:out value="${row.phrase_val}"/>
    </a>
    <br />
  </c:forEach>
```

CHAPTER 19

Generating Web Content from Query Results

19.0. Introduction

Information stored in a database is easily retrieved for use on the Web in a variety of ways. Query results can be displayed as unstructured paragraphs or as structured elements such as lists or tables. Query metadata can be useful when formatting query results, too, such as when generating an HTML table that displays a result set and uses its metadata to get the column headings for the table. These tasks combine statement processing with web scripting, and are primarily a matter of properly encoding special characters in the results (such as `&` or `<`) and adding the appropriate HTML tags for the types of elements you want to produce.

This chapter shows how to generate several types of web output from query results: paragraphs, lists, tables, hyperlinks, and navigation indexes (single- and multiple-page). The chapter also covers techniques for inserting binary data into your database and for retrieving and transferring that kind of information to clients. (It's easiest and most common to work with text for creating web pages from database content, but you can also use MySQL to service requests for binary data such as images, sounds, or PDF files.) You can also serve query results for download rather than for display.

The recipes here build on the techniques shown in Chapter 18 for generating web pages from scripts and encoding output for display. See that chapter if you need background in these topics.

Scripts to create tables used in this chapter are located in the *tables* directory of the `recipes` distribution. Scripts for the examples are located under the directories named for the web servers used to run them. For Perl, Ruby, PHP, and Python examples, look under the *apache* directory. Utility routines used by the example scripts are found in files located in the *lib* directory. (For information on configuring Apache so that scripts

run by it can find their library files, see Recipe 18.2.) For Java (JSP) examples, look under the *tomcat* directory; you should already have installed these in the process of setting up the `mcb` application context (see Recipe 18.3).

If a particular section has no example for a language in which you're interested, check the `recipes` distribution for implementations not shown here.

The scripts in this chapter are intended to be invoked from your browser after they have been installed, but you can invoke many of them (JSP pages excepted) from the command line to see the raw HTML they produce; see Recipe 18.2.

19.1. Displaying Query Results as Paragraphs

Problem

You want to display a query result as free text.

Solution

Display it within paragraph tags.

Discussion

Paragraphs are useful for displaying free text with no particular structure. Retrieve the text to be displayed, encode it to convert any special characters to the corresponding HTML entities, and wrap each paragraph within `<p>` and `</p>` tags. The following examples show how to produce paragraphs for a status display that includes the current date and time, the server version, and the default database name (if any). These values are available from the following query:

```
mysql> SELECT NOW(), VERSION(), DATABASE();
+---------------------+------------+------------+
| NOW()               | VERSION()  | DATABASE() |
+---------------------+------------+------------+
| 2013-12-22 11:29:50 | 5.6.16-log | cookbook   |
+---------------------+------------+------------+
```

One complication is that the `DATABASE()` result is `NULL` if there is no default database. The examples show how to handle this.

In Perl, the CGI.pm module provides a `p()` function that adds paragraph tags around the string you pass to it. `p()` does not HTML-encode its argument, so handle that by calling `escapeHTML()`:

```
($now, $version, $db) =
  $dbh->selectrow_array ("SELECT NOW(), VERSION(), DATABASE()");
$db = "NONE" unless defined ($db);
print p (escapeHTML ("Local time on the MySQL server is $now."));
```

```
print p (escapeHTML ("The server version is $version."));
print p (escapeHTML ("The default database is $db."));
```

In Ruby, use the `cgi` module `escapeHTML` method to encode the paragraph text, and then pass it to the `p` method to produce the paragraph tags:

```
(now, version, db) =
  dbh.select_one("SELECT NOW(), VERSION(), DATABASE()")
db = "NONE" if db.nil?
cgi = CGI.new("html4")
cgi.out {
  cgi.p { CGI.escapeHTML("Local time on the MySQL server is #{now}.") } +
  cgi.p { CGI.escapeHTML("The server version is #{version}.") } +
  cgi.p { CGI.escapeHTML("The default database is #{db}.") }
}
```

For languages without HTML-tag methods for the required elements, put `<p>` and `</p>` tags around the encoded paragraph text. PHP and Python are examples of this.

PHP:

```
$sth = $dbh->query ("SELECT NOW(), VERSION(), DATABASE()");
list ($now, $version, $db) = $sth->fetch (PDO::FETCH_NUM);
if ($db === NULL)
  $db = "NONE";
$para = "Local time on the MySQL server is $now.";
print ("<p>" . htmlspecialchars ($para) . "</p>");
$para = "The server version is $version.";
print ("<p>" . htmlspecialchars ($para) . "</p>");
$para = "The default database is $db.";
print ("<p>" . htmlspecialchars ($para) . "</p>");
```

Python:

```
cursor = conn.cursor()
cursor.execute("SELECT NOW(), VERSION(), DATABASE()")
(now, version, db) = cursor.fetchone()
cursor.close()
if db is None:
  db = 'NONE'
para = "Local time on the MySQL server is %s." % now
print("<p>%s</p>" % cgi.escape(para, 1))
para = "The server version is %s." % version
print("<p>%s</p>" % cgi.escape(para, 1))
para = "The default database is %s." % db
print("<p>%s</p>" % cgi.escape(para, 1))
```

In JSP, produce the paragraph display using `rowsByIndex` to access the result set row's columns by numeric index and `<c:out>` to encode and print the text:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT NOW(), VERSION(), DATABASE()
</sql:query>
<c:set var="row" value="${rs.rowsByIndex[0]}"/>
```

```
<c:set var="db" value="${row[2]}"/>
<c:if test="${empty db}">
  <c:set var="db" value="NONE"/>
</c:if>

<p>Local time on the server is <c:out value="${row[0]}"/>.</p>
<p>The server version is <c:out value="${row[1]}"/>.</p>
<p>The default database is <c:out value="${db}"/>.</p>
```

19.2. Displaying Query Results as Lists

Problem

You want to display a query result as a structured list of items.

Solution

There are several HTML list types. Write the list items within tags appropriate for the desired type of list.

Discussion

More structured than paragraphs and less structured than tables, lists provide a useful way to display a set of individual items. HTML provides several styles of lists, such as ordered lists, unordered lists, and definition lists. To nest lists, use list-within-list formatting.

Lists generally consist of opening and closing tags that enclose a set of items, each delimited by its own tags. List items correspond naturally to rows returned from a query, so generating an HTML list structure from within a program is a matter of encoding your query result, enclosing each row within the proper item tags, and adding the opening and closing list tags.

Two approaches to list generation are common: To print the tags as you process the result set, print the list opening tag, fetch and print each result set row as a list item, including the item tags, and print the list closing tag.

Alternatively, to process the list in memory, store the list items in an array, pass the array to a list-generation function that adds the appropriate tags, and print the result.

The examples that follow demonstrate both approaches.

Ordered lists

An ordered list consists of items that have a particular sequence. Browsers typically display ordered lists as a set of numbered items:

```
1. First item
2. Second item
3. Third item
```

You need not specify the item numbers because browsers add them automatically. An ordered list is enclosed within `<ol>` and `</ol>` tags, and contains items each enclosed within `<li>` and `</li>` tags:

```
<ol>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ol>
```

Suppose that an `ingredient` table contains numbered ingredients for a cooking recipe:

```
mysql> SELECT * FROM ingredient ORDER BY id;
+----+--------------------------------+
| id | item                           |
+----+--------------------------------+
|  1 | 3 cups flour                   |
|  2 | 1/2 cup raw ("unrefined") sugar |
|  3 | 3 eggs                         |
|  4 | pinch (< 1/16 teaspoon) salt   |
+----+--------------------------------+
```

The table contains an `id` column, but you need only fetch the text values in the proper order to display them as an ordered list because a browser adds item numbers itself. The items contain the special characters " and <, so HTML-encode them before adding the tags that convert the items to an HTML list. The result looks like this:

```
<ol>
  <li>3 cups flour</li>
  <li>1/2 cup raw (&amp;quot;unrefined&amp;quot;) sugar</li>
  <li>3 eggs</li>
  <li>pinch (&amp;lt; 1/16 teaspoon) salt</li>
</ol>
```

One way to create such list from a script is by printing the HTML as you fetch the rows of the result set. Here's how you might do so in a JSP page using the JSTL tag library:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT item FROM ingredient ORDER BY id
</sql:query>
<ol>
<c:forEach items="${rs.rows}" var="row">
  <li><c:out value="${row.item}"/></li>
</c:forEach>
</ol>
```

In PHP, perform the same operation like this:

```
$stmt = "SELECT item FROM ingredient ORDER BY id";
$sth = $dbh->query ($stmt);
```

```
print ("<ol>");
while (list ($item) = $sth->fetch (PDO::FETCH_NUM))
  print ("<li>" . htmlspecialchars ($item) . "</li>");
print ("</ol>");
```

The preceding examples generate HTML by interleaving row fetching and output generation. It's also possible to separate (decouple) the two operations: retrieve the data first, and then write the output. Queries tend to vary from list to list, but generating the list itself often is fairly stereotypical. If you put the list-generation code into a utility function, you can reuse it for different queries. The function must handle two operations: HTML-encoding the items (if they aren't already encoded), and adding the proper HTML tags. The following PHP function does this. It takes the list items as an array argument and returns the list as a string:

```
function make_ordered_list ($items, $encode = TRUE)
{
  $result = "";
  foreach ($items as $val)
  {
    if ($encode)
      $val = htmlspecialchars ($val);
    $result .= "<li>$val</li>";
  }
  return ("<ol>$result</ol>");
}
```

To use the utility function, fetch the data and print the HTML like so:

```
# fetch items for list
$stmt = "SELECT item FROM ingredient ORDER BY id";
$sth = $dbh->query ($stmt);
$items = $sth->fetchAll (PDO::FETCH_COLUMN, 0);

# generate HTML list
print (make_ordered_list ($items));
```

In Python, write the utility function like this:

```
def make_ordered_list(items, encode=True):
  result = ""
  for item in items:
    if item is None:  # handle possibility of NULL item
      item = ""
    # make sure item is a string, then encode if necessary
    item = str(item)
    if encode:
      item = cgi.escape(item, 1)
    result += "<li>" + item + "</li>"
  return "<ol>" + result + "</ol>"
```

And use it like this:

```
# fetch items for list
stmt = "SELECT item FROM ingredient ORDER BY id"
cursor = conn.cursor()
cursor.execute(stmt)
items = []
for (item,) in cursor:
  items.append(item)
cursor.close()

# generate HTML list
print(make_ordered_list(items))
```

The second argument to `make_ordered_list()` indicates whether it should perform HTML-encoding of the list items. The easiest thing is to let the function handle this for you (which is why the default is true). However, for a list of items that themselves include HTML tags, you wouldn't want the function to encode the special characters in those tags. For example, if the list items are hyperlinks, each contains `<a>` tags. To prevent these from being converted to `&lt;a&gt;`, pass `make_ordered_list()` a second argument that evaluates to false.

If your API provides functions to generate HTML structures, you need not write them yourself. That's the case for the Perl CGI.pm and Ruby `cgi` modules. In Perl, generate each item by invoking its `li()` function to add the opening and closing item tags, save the items in an array, and pass the array to `ol()` to add the opening and closing list tags:

```
my $stmt = "SELECT item FROM ingredient ORDER BY id";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
my @items;
while (my $ref = $sth->fetchrow_arrayref ())
{
  # handle possibility of NULL (undef) item
  my $item = defined ($ref->[0]) ? escapeHTML ($ref->[0]) : "";
  push (@items, li ($item));
}
print ol (@items);
```

The code converts `NULL` values (represented by `undef`) to the empty string is to avoid having Perl generate uninitialized-value warnings when run with warnings enabled. (The `ingredient` table doesn't actually contain any `NULL` values, but in the general case, you don't know that.)

The preceding example intertwines row fetching and HTML generation. To decouple item fetching from printing the HTML, first retrieve the items into an array, then pass the array by reference to `li()` and the result to `ol()`:

```
# fetch items for list
my $stmt = "SELECT item FROM ingredient ORDER BY id";
my $item_ref = $dbh->selectcol_arrayref ($stmt);

# generate HTML list, handling possibility of NULL (undef) items
```

```
$item_ref = [ map { defined ($_) ? escapeHTML ($_) : "" } @{$item_ref} ];
print ol (li ($item_ref));
```

Note two things about the `li()` function:

- It performs no HTML-encoding; you must do that yourself.
- It can handle a single value or an array of values. If you pass an array, pass it by reference so that `li()` adds `<li>` and `</li>` tags to each array element, concatenates them, and returns the resulting string. If you pass the array itself rather than a reference, `li()` first concatenates the items, then adds a single set of tags around the result, which is usually not what you want. This behavior is shared by several other CGI.pm functions that can operate on single or multiple values. For example, the table data `td()` function adds a single set of `<td>` and `</td>` tags if you pass it a scalar or list. If you pass a list reference, it adds the tags to each item in the list.

The Ruby equivalent of the previous example looks like this:

```
# fetch items for list
stmt = "SELECT item FROM ingredient ORDER BY id"
items = dbh.select_all(stmt)

list = cgi.ol {
  items.collect { |item| cgi.li { CGI.escapeHTML(item.to_s) } }
}
```

Should You Intertwine or Decouple Row Fetching and HTML Generation?

You may be able to get a script working most quickly by writing code that prints HTML from query rows as you fetch them. There are, however, several advantages to separating data retrieval from output production. The most obvious ones are that by using a utility function to generate the HTML, you have to write the function only once, and you can share it among scripts. There are other benefits as well:

- Functions that generate HTML structures can be used with data obtained from other sources, not just from a database.
- The decoupled approach enables you to more easily construct page content in memory, then print it when you're ready. For building pages that consist of several components, this gives you more latitude to create them in the order that's most convenient. (On the other hand, with very large result sets, this approach can entail considerable memory use.)
- Decoupling row fetching and output generation provides more flexibility in the types of output you produce. To generate an unordered list rather than an ordered list, just call a different output function; the data collection phase need not change. This is true even if you decide to use a different output language (XML rather than

HTML, for example). You still need only a different output function; data collection remains unchanged.

- By prefetching the list items, you can make adaptive decisions about what type of list to create. Although we are not yet to the point of discussing web forms, they make heavy use of their own kinds of lists. In that context, having items in hand before generating an HTML structure from them can be useful for choosing the list type based on the list size. For example, you can display a set of radio buttons if the number of items is small, or a pop-up menu or scrolling list if the number is large.

Unordered lists

An unordered list is like an ordered list except that browsers display all items with the same marker character, such as a bullet:

- First item
- Second item
- Third item

"Unordered" refers to the fact that the marker character provides no sequence information. You can of course display the items in any order you choose. The HTML tags for an unordered list are the same as for an ordered list except that the opening and closing tags are `<ul>` and `</ul>` rather than `<ol>` and `</ol>`:

```
<ul>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ul>
```

For APIs in which you print the tags directly, use the same procedure as for ordered lists, but print `<ul>` and `</ul>` instead of `<ol>` and `</ol>`. Here is an example in JSP:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT item FROM ingredient ORDER BY id
</sql:query>
<ul>
<c:forEach items="${rs.rows}" var="row">
  <li><c:out value="${row.item}"/></li>
</c:forEach>
</ul>
```

For APIs that provide tag-generating methods, call a different method to produce the outer tags. For example, in Perl, create an unordered list by calling the CGI.pm `ul()` function rather than `ol()`.

To write a utility function for unordered lists, it's easily derived from a function that generates ordered lists because they differ only in the opening and closing list tags.

Definition lists

A definition list consists of two-part items, each including a term and a definition. "Term" and "definition" have loose meanings because you can display any kind of information you want. The following `doremi` table associates the name of each note in a musical scale with a mnemonic phrase for remembering it:

```
mysql> SELECT id, note, mnemonic FROM doremi ORDER BY id;
+----+------+---------------------------+
| id | note | mnemonic                  |
+----+------+---------------------------+
|  1 | do   | A deer, a female deer     |
|  2 | re   | A drop of golden sun      |
|  3 | mi   | A name I call myself      |
|  4 | fa   | A long, long way to run   |
|  5 | so   | A needle pulling thread   |
|  6 | la   | A note to follow so       |
|  7 | ti   | A drink with jam and bread |
+----+------+---------------------------+
```

The mnemonics aren't exactly what you'd call "definitions"; nevertheless, the `note` and `mnemonic` columns can be displayed as a definition list:

```
do
    A deer, a female deer
re
    A drop of golden sun
mi
    A name I call myself
fa
    A long, long way to run
so
    A needle pulling thread
la
    A note to follow so
ti
    A drink with jam and bread
```

A definition list begins and ends with `<dl>` and `</dl>` tags. Each item has a term enclosed within `<dt>` and `</dt>` tags and a definition enclosed within `<dd>` and `</dd>` tags:

```
<dl>
  <dt>do</dt> <dd>A deer, a female deer</dd>
  <dt>re</dt> <dd>A drop of golden sun</dd>
  <dt>mi</dt> <dd>A name I call myself</dd>
  <dt>fa</dt> <dd>A long, long way to run</dd>
  <dt>so</dt> <dd>A needle pulling thread</dd>
  <dt>la</dt> <dd>A note to follow so</dd>
  <dt>ti</dt> <dd>A drink with jam and bread</dd>
</dl>
```

In a JSP page, generate the definition list like this:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT note, mnemonic FROM doremi ORDER BY note
</sql:query>
<dl>
<c:forEach items="${rs.rows}" var="row">
  <dt><c:out value="${row.note}"/></dt>
  <dd><c:out value="${row.mnemonic}"/></dd>
</c:forEach>
</dl>
```

Alternatively, fetch the data and then pass it to a utility function that takes arrays of terms and definitions and returns the list as a string. Here is an example in PHP:

```
# fetch items for list
$stmt = "SELECT note, mnemonic FROM doremi ORDER BY id";
$sth = $dbh->query ($stmt);
$terms = array ();
$defs = array ();
while (list ($note, $mnemonic) = $sth->fetch (PDO::FETCH_NUM))
{
  $terms[] = $note;
  $defs[] = $mnemonic;
}

# generate HTML list
print (make_definition_list ($terms, $defs));
```

The `make_definition_list()` function looks like this:

```
function make_definition_list ($terms, $definitions, $encode = TRUE)
{
  $result = "";
  $n = count ($terms);
  for ($i = 0; $i < $n; $i++)
  {
    $term = $terms[$i];
    $definition = $definitions[$i];
    if ($encode)
    {
      $term = htmlspecialchars ($term);
      $definition = htmlspecialchars ($definition);
    }
    $result .= "<dt>$term</dt><dd>$definition</dd>";
  }
  return ("<dl>$result</dl>");
}
```

In Ruby, use the `dt` and `dd` methods to create list item contents, then pass the result to the `dl` method to add the outermost list tags:

```
stmt = "SELECT note, mnemonic FROM doremi ORDER BY id"
list = ""
dbh.execute(stmt) do |sth|
  sth.fetch do |row|
```

```
    list << cgi.dt { CGI.escapeHTML(row["note"].to_s) }
    list << cgi.dd { CGI.escapeHTML(row["mnemonic"].to_s) }
  end
end
list = cgi.dl { list }
```

Here is another example (in Perl). Each term is a database name, and the corresponding definition indicates how many tables are in the database. The numbers are obtained from `INFORMATION_SCHEMA` using a query that counts the number of tables in each database. Create the terms and definitions by invoking `dt()` and `dd()`, save them in an array, and pass the array to `dl()`:

```
# count number of tables per database
my $sth = $dbh->prepare ("SELECT TABLE_SCHEMA, COUNT(TABLE_NAME)
                          FROM INFORMATION_SCHEMA.TABLES
                          GROUP BY TABLE_SCHEMA");
$sth->execute ();
my @items;
while (my ($db_name, $tbl_count) = $sth->fetchrow_array ())
{
  push (@items, dt (escapeHTML ($db_name)));
  push (@items, dd (escapeHTML ($tbl_count . " tables")));
}
print dl (@items);
```

The counts indicate the number of tables accessible to the MySQL user account that the script uses when it connects to the MySQL server. Databases or tables not accessible to that account are not included.

Nested lists

Some information is most easily understood when presented as a list of lists. The following example displays state names as a definition list, grouped by initial letter of the names. For each item in the list, the term is the initial letter, and the definition is an unordered list of the state names beginning with that letter:

```
A
  • Alabama
  • Alaska
  • Arizona
  • Arkansas
C
  • California
  • Colorado
  • Connecticut
D
  • Delaware
…
```

One way to produce such a list (in Perl) is to retrieve all the information using a single query, marching through the result set, and beginning a new list item each time you reach a new letter:

```
my $sth = $dbh->prepare ("SELECT name FROM states ORDER BY name");
$sth->execute ();
my @items;
my @names;
my $cur_ltr = "";
while (my ($name) = $sth->fetchrow_array ())
{
  my $ltr = uc (substr ($name, 0, 1));  # initial letter of name
  if ($cur_ltr ne $ltr)                 # beginning a new letter?
  {
    if (@names)       # any stored-up names from previous letter?
    {
      # for each definition list item, the initial letter is
      # the term, and the list of states is the definition
      push (@items, dt ($cur_ltr));
      push (@items, dd (ul (li (\@names))));
    }
    @names = ();
    $cur_ltr = $ltr;
  }
  push (@names, escapeHTML ($name));
}
if (@names)           # any remaining names from final letter?
{
  push (@items, dt ($cur_ltr));
  push (@items, dd (ul (li (\@names))));
}
print dl (@items);
```

Another approach uses the same query but separates the data-collection and HTML-generation phases:

```
# collect state names and associate each with the proper
# initial-letter list
my $sth = $dbh->prepare ("SELECT name FROM states ORDER BY name");
$sth->execute ();
my %ltr;
while (my ($name) = $sth->fetchrow_array ())
{
  my $ltr = uc (substr ($name, 0, 1));  # initial letter of name
  # initialize letter list to empty array if this is
  # first state for it, then add state to array
  $ltr{$ltr} = [] unless exists ($ltr{$ltr});
  push (@{$ltr{$ltr}}, $name);
}

# generate the output lists
my @items;
```

```
foreach my $ltr (sort (keys (%ltr)))
{
  # encode list of state names for this letter, generate unordered list
  my $ul_str = ul (li ([ map { escapeHTML ($_) } @{$ltr{$ltr}} ]));
  push (@items, dt ($ltr), dd ($ul_str));
}
print dl (@items);
```

For another application of nested lists, see Recipe 19.5.

19.3. Displaying Query Results as Tables

Problem

You want to display a query result as an HTML table.

Solution

Use each row of the result as a table row. To present an initial row of column labels, supply your own or use the query metadata to obtain the column names.

Discussion

HTML tables are useful for presenting highly structured output. They're popular for displaying the results of queries that consist of rows and columns due to the natural conceptual correspondence between HTML tables and database tables or query results. In addition, you can obtain column headers for the table by accessing the query metadata (see Recipe 10.2). An HTML table has this basic structure:

- The table begins and ends with `<table>` and `</table>` tags and encloses a set of rows.
- Each row begins and ends with `<tr>` and `</tr>` tags and encloses a set of cells.
- Tags for header cells are `<th>` and `</th>`. Tags for data cells are `<td>` and `</td>`. (Typically, browsers display header cells using boldface or other emphasis.)
- Tags may include attributes. For example, to put a border around each cell, add a `border="1"` attribute to the `<table>` tag. To right-justify a table cell, add an `align="right"` attribute to the `<td>` tag.

Suppose that you want to display the contents of your CD collection:

```
mysql> SELECT year, artist, title FROM cd ORDER BY artist, year;
+------+----------------+------------------+
| year | artist         | title            |
+------+----------------+------------------+
| 2002 | Aradhna        | Marga Darshan    |
| 1999 | Charlie Peacock | Kingdom Come     |
```

```
| 2008 | Children 18:3   | Children 18:3     |
| 2004 | Dave Bainbridge | Veil of Gossamer  |
| 1990 | Iona            | Iona              |
| 2011 | Iona            | Another Realm     |
| 2005 | Neal Morse      | ?                 |
| 2013 | Red             | Release the Panic |
| 1982 | Undercover      | Undercover        |
+------+-----------------+-------------------+
```

To display this query result as a bordered HTML table, produce output that looks something like this:

```
<table border="1">
  <tr>
    <th>Year</th>
    <th>Artist</th>
    <th>Title</th>
  </tr>
  <tr>
    <td>2002</td>
    <td>Aradhna</td>
    <td>Marga Darshan</td>
  </tr>
  <tr>
    <td>1999</td>
    <td>Charlie Peacock</td>
    <td>Kingdom Come</td>
  </tr>
  ... other rows here ...
  <tr>
    <td>1982</td>
    <td>Undercover</td>
    <td>Undercover</td>
  </tr>
</table>
```

To convert the results of a query to an HTML table, wrap each value from a given result set row in cell tags, each row in row tags, and the entire set of rows in table tags. A JSP page might produce an HTML table from the cd table query like this:

```
<table border="1">
  <tr>
    <th>Year</th>
    <th>Artist</th>
    <th>Title</th>
  </tr>
<sql:query dataSource="${conn}" var="rs">
  SELECT year, artist, title FROM cd ORDER BY artist, year
</sql:query>
<c:forEach items="${rs.rows}" var="row">
  <tr>
    <td><c:out value="${row.year}"/></td>
    <td><c:out value="${row.artist}"/></td>
```

```
    <td><c:out value="${row.title}"/></td>
  </tr>
</c:forEach>

</table>
```

In Perl scripts, the CGI.pm functions `table()`, `tr()`, `td()`, and `th()` produce the table, row, data cell, and header cell elements. (Special case: To avoid a conflict with the built-in Perl `tr` character-transliteration function, invoke the `tr()` function that generates a table row as `Tr()`.) To display the contents of the `cd` table as an HTML table, do this:

```
my $sth = $dbh->prepare ("SELECT year, artist, title
                          FROM cd ORDER BY artist, year");
$sth->execute ();
my @rows;
push (@rows, Tr (th ("Year"), th ("Artist"), th ("Title")));
while (my ($year, $artist, $title) = $sth->fetchrow_array ())
{
  push (@rows, Tr (
                td (escapeHTML ($year)),
                td (escapeHTML ($artist)),
                td (escapeHTML ($title))
              ));
}
print table ({-border => "1"}, @rows);
```

Sometimes a table is easier to read if the rows use alternating colors, particularly if its cells don't include borders. To do this, add a `style` attribute that sets the background color to each `<th>` and `<td>` tag, and alternate the color value for each row. This is easy with a variable that toggles between two values. The following example alternates the `$color` variable between `silver` and `white`:

```
my $sth = $dbh->prepare ("SELECT year, artist, title
                          FROM cd ORDER BY artist, year");
$sth->execute ();
my $color = "silver";   # row-color variable
my $style = "background-color:$color";
my @rows;
push (@rows, Tr (
              th ({-style => $style}, "Year"),
              th ({-style => $style}, "Artist"),
              th ({-style => $style}, "Title")
            ));
while (my ($year, $artist, $title) = $sth->fetchrow_array ())
{
  # toggle the row-color variable
  $color = ($color eq "silver" ? "white" : "silver");
  $style = "background-color:$color";
  push (@rows, Tr (
                td ({-style => $style}, escapeHTML ($year)),
                td ({-style => $style}, escapeHTML ($artist)),
                td ({-style => $style}, escapeHTML ($title))
```

```
                ));
}
print table ({-border => "1"}, @rows);
```

The preceding table-generation examples hardwire the column headings into the code, as well as knowledge about the number of columns. With a little effort, you can write a more general function that takes a database handle and an arbitrary statement, executes the statement, and returns its result as an HTML table. The function gets the column labels from the statement metadata. To produce labels that differ from the table column names, specify column aliases in the statement:

```
my $tbl_str = make_table_from_query (
                $dbh,
                "SELECT
                   year AS Year, artist AS Artist, title AS Title
                 FROM cd
                 ORDER BY artist, year"
              );
print $tbl_str;
```

Any kind of statement that returns a result set can be passed to this function. You could, for example, use it to construct an HTML table from the result of a `CHECK TABLE` statement, which returns a result set that indicates the outcome of the check operation.

What does the `make_table_from_query()` function look like? Here's a Perl implementation:

```
sub make_table_from_query
{
# db handle, query string, parameters to be bound to placeholders (if any)
my ($dbh, $stmt, @param) = @_;

  my $sth = $dbh->prepare ($stmt);
  $sth->execute (@param);
  my @rows;
  # use column names for cells in the header row
  push (@rows, Tr (th ([ map { escapeHTML ($_) } @{$sth->{NAME}} ])));
  # fetch each data row
  while (my $row_ref = $sth->fetchrow_arrayref ())
  {
    # encode cell values, avoiding warnings for undefined
    # values and using   for empty cells
    my @val = map {
                defined ($_) && $_ !~ /^\s*$/ ? escapeHTML ($_) : " "
              } @{$row_ref};
    my $row_str;
    for (my $i = 0; $i < @val; $i++)
    {
      # right-justify numeric columns
      if ($sth->{mysql_is_num}->[$i])
      {
        $row_str .= td ({-align => "right"}, $val[$i]);
```

```
        }
        else
        {
          $row_str .= td ($val[$i]);
        }
      }
      push (@rows, Tr ($row_str));
    }
    return table ({-border => "1"}, @rows);
}
```

`make_table_from_query()` does some extra work to right-justify numeric columns so that the values line up better. It also enables you to pass values to be bound to placeholders in the statement; specify them after the statement string:

```
my $tbl_str = make_table_from_query (
                $dbh,
                "SELECT
                   year AS Year, artist AS Artist, title AS Title
                 FROM cd
                 WHERE year < ?
                 ORDER BY artist, year",
                1995
              );
print $tbl_str;
```

To display a table in such a way that the user can click any column heading to sort the table's contents by that column, see Recipe 20.11.

The   Trick for Empty Table Cells

A display problem sometimes occurs for HTML tables that include borders around cells: when a table cell is empty or contains only whitespace, some browsers show no border around the cell. This makes the table look irregular. To avoid this problem, the `make_table_from_query()` function puts a nonbreaking space (` `) into cells that would otherwise be empty, so that borders for them display properly.

19.4. Displaying Query Results as Hyperlinks

Problem

You want to create clickable hyperlinks from database content.

Solution

Add the proper tags to the content to generate anchor elements.

Discussion

The examples in the preceding sections generate static text, but database content also is useful for creating hyperlinks. Website URLs or email addresses stored in a table are easily converted to active links in web pages. You need only properly encode the information and add the appropriate HTML tags.

Suppose that a table named `book_vendor` contains bookseller and publisher names and websites:

```
mysql> SELECT * FROM book_vendor ORDER BY name;
+---------------+-----------------------+
| name          | website               |
+---------------+-----------------------+
| Amazon.com    | www.amazon.com        |
| Barnes & Noble | www.barnesandnoble.com |
| O'Reilly Media | www.oreilly.com       |
+---------------+-----------------------+
```

This table readily lends itself to the creation of hyperlinked text. To produce a hyperlink from a row, add the `http://` protocol designator to the `website` value, use the result as the `href` attribute for an `<a>` anchor tag, and use the `name` value in the body of the tag to serve as the link label. Here is the result for the Barnes & Noble row:

```
<a href="http://www.barnesandnoble.com">Barnes &amp;amp; Noble</a>
```

JSP code to produce an unordered list of hyperlinks from the table contents looks like this:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT name, website FROM book_vendor ORDER BY name
</sql:query>

<ul>
<c:forEach items="${rs.rows}" var="row">
  <li>
    <a href="http://<c:out value="${row.website}"/>">
      <c:out value="${row.name}"/></a>
  </li>
</c:forEach>
</ul>
```

When displayed in a web page, each vendor name in the list becomes an active link that can be selected to visit the vendor's website. In Python, the equivalent operation looks like this:

```
stmt = "SELECT name, website FROM book_vendor ORDER BY name"
cursor = conn.cursor()
cursor.execute(stmt)
items = []
for (name, website) in cursor:
  items.append('<a href="http://%s">%s</a>'
```

```
                % (urllib.quote(website), cgi.escape(name, 1)))
cursor.close()

# print items, but don't encode them; they're already encoded
print(make_unordered_list(items, False))
```

CGI.pm-based Perl scripts produce hyperlinks with the `a()` function:

```
a ({-href => "url-value"}, "link label")
```

Use the function to produce the vendor link list like this:

```
my $stmt = "SELECT name, website FROM book_vendor ORDER BY name";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
my @items;
while (my ($name, $website) = $sth->fetchrow_array ())
{
  push (@items, a ({-href => "http://$website"}, escapeHTML ($name)));
}
print ul (li (\@items));
```

Ruby scripts use the `cgi` module `a` method to produce hyperlinks:

```
stmt = "SELECT name, website FROM book_vendor ORDER BY name"
list = ""
dbh.execute(stmt) do |sth|
  sth.fetch do |row|
    list << cgi.li {
              cgi.a("href" => "http://#{row[1]}") {
                CGI.escapeHTML(row[0].to_s)
              }
            }
  end
end
list = cgi.ul { list }
```

Generating links using email addresses is another common web programming task. Assume that a table named `newsstaff` lists the department, name, and (if known) email address for the news anchors and reporters employed by a television station, WRRR:

```
mysql> SELECT * FROM newsstaff;
+------------------+----------------+-------------------------+
| department       | name           | email                   |
+------------------+----------------+-------------------------+
| Sports           | Becky Winthrop | bwinthrop@wrrr-news.com |
| Weather          | Bill Hagburg   | bhagburg@wrrr-news.com  |
| Local News       | Frieda Stevens | NULL                    |
| Local Government | Rex Conex      | rconex@wrrr-news.com    |
| Current Events   | Xavier Ng      | xng@wrrr-news.com       |
+------------------+----------------+-------------------------+
```

From this you want to produce an online directory containing email links to all personnel, so that site visitors can send mail to any staff member. For example, a row for

the sports reporter Becky Winthrop with an email address of `bwinthrop@wrrr-news.com` becomes an entry in the listing that looks like this:

```
Sports: <a href="mailto:bwinthrop@wrrr-news.com">Becky Winthrop</a>
```

It's easy to use the table's contents to produce such a directory. First, put the code to generate an email link into a helper function because that operation is likely to be useful in multiple scripts. In Perl, the function looks like this:

```
sub make_email_link
{
my ($name, $addr) = @_;

  $name = escapeHTML ($name);
  # return name as static text if address is undef or empty
  return $name if !defined ($addr) || $addr eq "";
  # return a hyperlink otherwise
  return a ({-href => "mailto:$addr"}, $name);
}
```

The function handles instances where the person has no email address by returning just the name as static text. To use the function, write a loop that pulls out names and addresses and displays each email link preceded by the staff member's department:

```
my $stmt = "SELECT department, name, email FROM newsstaff
            ORDER BY department, name";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();
my @items;
while (my ($dept, $name, $email) = $sth->fetchrow_array ())
{
  push (@items,
        escapeHTML ($dept) . ": " . make_email_link ($name, $email));
}
print ul (li (\@items));
```

Equivalent email link generator functions for Ruby, PHP, and Python are similar.

For a JSP page, produce the `newsstaff` listing as follows:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT department, name, email
  FROM newsstaff
  ORDER BY department, name
</sql:query>

<ul>
<c:forEach items="${rs.rows}" var="row">
  <li>
    <c:out value="${row.department}"/>:
    <c:set var="name" value="${row.name}"/>
    <c:set var="email" value="${row.email}"/>
    <c:choose>
```

```
        <%-- null or empty value test --%>
        <c:when test="${empty email}">
          <c:out value="${name}"/>
        </c:when>
        <c:otherwise>
          <a href="mailto:<c:out value="${email}"/>">
            <c:out value="${name}"/></a>
        </c:otherwise>
      </c:choose>
    </li>
  </c:forEach>
  </ul>
```

19.5. Creating Navigation Indexes from Database Content

Problem

A list of items in a web page is long. You want to make it easier for users to move around in the list.

Solution

Create a navigation index containing links to different sections of the list.

Discussion

It's easy to display a list in a web page (see Recipe 19.2), but if the list contains a lot of items, the page becomes quite long. It's often useful to break a lengthy list into sections and provide a navigation index in the form of hyperlinks that enable users to reach sections of the list directly without scrolling the page. For example, if you retrieve rows from a table and display them grouped into sections, you can include an index that lets the user jump to any section. The same idea applies to multiple-page displays as well, using a navigation index in each page that enables users to reach any other page easily.

This recipe provides examples to illustrate both techniques, using the `kjv` table introduced in Recipe 5.12:

- A single-page display that lists all verses in all chapters of the book of Esther. The list is broken into 10 sections (one per chapter), with a navigation index that has links pointing to the beginning of each section.
- A multiple-page display consisting of pages that each show the verses from a single chapter of Esther, and a list of links to pages for each of the other chapters. These links enable any page to be reached easily from any other.

Creating a single-page navigation index

This example displays all verses in Esther in a single page, with verses grouped into sections by chapter. To display the page so that each section contains a navigation marker, place an `<a name>` anchor element before each chapter's verses:

```
<a name="1">Chapter 1</a>
  ... list of verses in chapter 1...
<a name="2">Chapter 2</a>
  ... list of verses in chapter 2...
<a name="3">Chapter 3</a>
  ... list of verses in chapter 3...
…
```

That generates a list that includes a set of markers named `1`, `2`, `3`, and so forth. To construct the navigation index, build a set of hyperlinks, each of which points to one of the `name` markers:

```
<a href="#1">Chapter 1</a>
<a href="#2">Chapter 2</a>
<a href="#3">Chapter 3</a>
…
```

The `#` in each `href` attribute signifies that the link points to a location within the same page. For example, `href="#3"` points to the anchor with the `name="3"` attribute.

To implement this kind of navigation index, use one of these approaches:

- Retrieve the verse rows into memory and determine from them the entries needed in the navigation index. Then print both the index and verse list.
- Figure out all the applicable anchors in advance and construct the index first. This statement determines the list of chapter numbers:

  ```
  SELECT DISTINCT cnum FROM kjv WHERE bname = 'Esther' ORDER BY cnum;
  ```

  Use the query result to build the navigation index, then fetch the verses for the chapters later to create the page sections to which the index entries point.

Here's a script, *esther1.pl*, that uses the first approach. It's an adaptation of one of the nested-list examples shown in Recipe 19.2:

```
#!/usr/bin/perl
# esther1.pl: display the book of Esther in a single page,
# with navigation index

use strict;
use warnings;
use CGI qw(:standard escape escapeHTML);
use Cookbook;

my $title = "The Book of Esther";
```

```
my $page = header ()
           . start_html (-title => $title)
           . h3 ($title);

my $dbh = Cookbook::connect ();

# Retrieve verses from the book of Esther and associate each one with the
# list of verses for the chapter it belongs to.

my $sth = $dbh->prepare ("SELECT cnum, vnum, vtext FROM kjv
                          WHERE bname = 'Esther'
                          ORDER BY cnum, vnum");
$sth->execute ();
my %verses;
while (my ($cnum, $vnum, $vtext) = $sth->fetchrow_array ())
{
  # Initialize chapter's verse list to empty array if this is
  # first verse for it, then add verse number/text to array.
  $verses{$cnum} = [] unless exists ($verses{$cnum});
  push (@{$verses{$cnum}}, p (escapeHTML ("$vnum. $vtext")));
}

# Determine all chapter numbers and use them to construct a navigation
# index.  These are links of the form <a href="#num>Chapter num</a>, where
# num is a chapter number and '#' signifies a within-page link.  No URL-
# or HTML-encoding is done here (the text displayed here doesn't need
# it).  Make sure to sort chapter numbers numerically (use { a <=> b }).
# Separate links by nonbreaking spaces.

my $nav_index;
foreach my $cnum (sort { $a <=> $b } keys (%verses))
{
  $nav_index .= " " if $nav_index;
  $nav_index .= a ({-href => "#$cnum"}, "Chapter $cnum");
}

# Display list of verses for each chapter.  Precede each section by a
# label that shows the chapter number and a copy of the navigation index.

foreach my $cnum (sort { $a <=> $b } keys (%verses))
{
  # add an <a name> anchor for this section of the chapter display
  $page .= p (a ({-name => $cnum}, font ({-size => "+2"}, "Chapter $cnum"))
           . br ()
           . $nav_index);
  $page .= join ("", @{$verses{$cnum}});  # add array of verses for chapter
}

$dbh->disconnect ();

$page .= end_html ();
```

```
print $page;
```

Creating a multiple-page navigation index

This example shows a Perl script, *esther2.pl*, that is capable of generating any of several pages, all based on the verses in the book of Esther stored in the `kjv` table. The initial page displays a list of the chapters in the book, along with the verses from chapter 1. Each item in the chapter list is a hyperlink that reinvokes the script to display the list of verses in one of the other chapters. Because the script is responsible for generating multiple pages, it must be able to determine which page to display each time it runs. To make that possible, the script examines its own URL for a `chapter` parameter that indicates the number of the chapter to display.

The URL to request the initial page looks like this:

```
http://localhost/cgi-bin/esther2.pl
```

The links to individual chapter pages have the following form, where *cnum* is a chapter number:

```
http://localhost/cgi-bin/esther2.pl?chapter=cnum
```

esther2.pl uses the CGI.pm `param()` function to obtain the `chapter` parameter value, defaulting to 1 if the chapter is missing or not integer-valued:

```
my $cnum = param ("chapter");
# Missing or malformed chapter; default to chapter 1.
$cnum = 1 if !defined ($cnum) || $cnum !~ /^\d+$/;
```

Here is the entire *esther2.pl* script:

```
#!/usr/bin/perl
# esther2.pl: display the book of Esther over multiple pages,
# one page per chapter, with navigation index

use strict;
use warnings;
use CGI qw(:standard escape escapeHTML);
use Cookbook;

# Construct navigation index as a list of links to the pages for each
# chapter in the the book of Esther.  Labels are of the form "Chapter
# n"; the chapter numbers are incorporated into the links as chapter=num
# parameters

# $dbh is the database handle, $cnum is the number of the chapter for
# which information is currently being displayed.  The label in the
# chapter list corresponding to this number is displayed as static
# text; the others are displayed as hyperlinks to the other chapter
# pages.
```

```
# No encoding is done because the chapter numbers are digits and don't
# need it.

sub get_chapter_list
{
my ($dbh, $cnum) = @_;

  my $nav_index;
  my $ref = $dbh->selectcol_arrayref (
                      "SELECT DISTINCT cnum FROM kjv
                       WHERE bname = 'Esther' ORDER BY cnum"
                );
  foreach my $cur_cnum (@{$ref})
  {
    my $link = url () . "?chapter=$cur_cnum";
    my $label = "Chapter $cur_cnum";
    $nav_index .= br () if $nav_index;      # separate entries by <br>
    # use static bold text if entry is for current chapter,
    # use a hyperlink otherwise
    $nav_index .= ($cur_cnum == $cnum
                   ? strong ($label)
                   : a ({-href => $link}, $label));
  }
  return $nav_index;
}

# Get the list of verses for a given chapter.  If there are none, the
# chapter number was invalid, but handle that case sensibly.

sub get_verses
{
my ($dbh, $cnum) = @_;

  my $ref = $dbh->selectall_arrayref (
                      "SELECT vnum, vtext FROM kjv
                       WHERE bname = 'Esther' AND cnum = ?",
                      undef, $cnum);
  my $verses = "";
  foreach my $row_ref (@{$ref})
  {
    $verses .= p (escapeHTML ("$row_ref->[0]. $row_ref->[1]"));
  }
  return $verses eq ""     # no verses?
         ? p ("No verses in chapter $cnum were found.")
         : p ("Chapter $cnum:") . $verses;
}

my $title = "The Book of Esther";

my $page = header () . start_html (-title => $title);

my $dbh = Cookbook::connect ();
```

```
my $cnum = param ("chapter");
# Missing or malformed chapter; default to chapter 1.
$cnum = 1 if !defined ($cnum) || $cnum !~ /^\d+$/;

# Arrange the page panels as a one-row, three-cell table:
# Left panel: List of chapters as hyperlinks (except for current chapter
# as bold text)
# Middle panel: Spacer
# Right panel: List of current chapter's verses

$page .= table (Tr (
                  td ({-valign => "top", -width => "15%"},
                      get_chapter_list ($dbh, $cnum)),
                  td ({-valign => "top", -width => "5%"}, " "),
                  td ({-valign => "top", -width => "80%"},
                      p (strong ($title)) . get_verses ($dbh, $cnum))
                ));

$dbh->disconnect ();

$page .= end_html ();

print $page;
```

See Also

esther2.pl examines its execution environment using the `param()` function. Recipe 20.5 further discusses web script parameter processing.

Recipe 20.10 discusses another navigation problem: how to split display of a result set across multiple pages and create previous-page and next-page links.

19.6. Storing Images or Other Binary Data

Problem

You want to store images in MySQL.

Solution

That's not difficult, provided that you take the proper precautions for encoding image data.

Discussion

Websites are not limited to displaying text. They also serve various forms of binary data such as images, music files, PDF documents, and so forth. Images are a common kind

of binary data, and because image storage is a natural application for a database, a very common question is "How do I store images in MySQL?" Many people answer this question by saying, "Don't do it!" and some of the reasons are discussed in the following sidebar. Because it's important to know how to work with binary data, this section does show how to store images in MySQL. Nevertheless, in recognition that that may not always be the best thing to do, the section also shows how to store images in the filesystem.

Although the discussion here is phrased in terms of working with images, the principles apply to any kind of binary data, such as PDF files or compressed text. In fact, they apply to any kind of data at all, including text. People tend to think of images as special somehow, but they're not.

One reason that image storage confuses people more often than does storing other types of information like text strings or numbers is that it's difficult to type in an image value manually. For example, you can easily use *mysql* to enter an `INSERT` statement to store a number like `3.48` or a string like `Je voudrais une bicyclette rouge`, but images contain binary data and it's not easy to refer to them by value. So you must do something else. Your options are:

- Use the `LOAD_FILE()` function.
- Write a program that reads in the image file and constructs the proper `INSERT` statement for you.

Either way, when you store images in the database, use a binary string data type such as a `BLOB`, not a character string type.

Should You Store Images in Your Database?

Deciding where to store images involves trade-offs. There are advantages and disadvantages regardless of whether you store images in the database or in the filesystem:

- Storing images in a database table bloats the table. With a lot of images, you're more likely to approach any limits your operating system places on table size. On the other hand, if you store images in the filesystem, directory lookups may become slow.
- Using a database centralizes storage for images that are used across multiple web servers on different hosts. Images stored in the filesystem must be stored locally on the web server host, so in a multiple-host situation, you must replicate the set of images to the filesystem of each host. If you store the images in MySQL, only one copy of the images is required because each web server can get the images from the same database server.

- Images stored in the filesystem constitute, in essence, a foreign key. Image manipulation requires two operations: one in the database and one in the filesystem. This in turn means that if you require transactional behavior, it's more difficult to implement—not only do you have two operations, but they take place in different domains. Storing images in the database is simpler because adding, updating, or removing an image requires only a single-row operation. It becomes unnecessary to make sure the image table and the filesystem remain in synchrony.
- It can be faster to serve images over the Web from the filesystem than from the database because the web server itself opens the file, reads it, and writes it to the client. Images stored in the database must be read and written twice. First, the MySQL server reads the image from the database and writes it to your web script. Then the script reads the image and writes it to the client.
- Images stored in the filesystem can be referred to directly in web pages by means of `<img>` tag links that point to the image files. Images stored in MySQL must be served by a script that retrieves an image and sends it to the client. However, even if images are stored in the filesystem and accessible to the web server, you might still want to serve them through a script. This would be appropriate if you must account for the number of times you serve each image (such as for banner ad displays where you charge customers by the number of ad impressions) or if you want to select an image at request time (such as when you pick an ad at random).

Storing images with LOAD_FILE()

The `LOAD_FILE()` function takes an argument indicating a file to be read and stored in the database. For example, to load an image stored in */tmp/myimage.png* into a table, do this:

```
INSERT INTO mytbl (image_data) VALUES(LOAD_FILE('/tmp/myimage.png'));
```

To load images with `LOAD_FILE()`, these requirements must be satisfied:

- The image file must be located on the MySQL server host.
- The file must be readable by the server.
- You must have the `FILE` privilege.

These constraints mean that `LOAD_FILE()` is available only to some MySQL users.

Storing images using a script

If `LOAD_FILE()` is not an option, or you don't want to use it, you can use a short program to load your images. The program should either read the contents of an image file and create a row that contains the image data, or create a row that indicates where in the filesystem the image file is located. If you elect to store the image in MySQL, include the image data in the row-creation statement the same way as any other kind of data.

That is, you either use a placeholder and bind the data value to it, or else encode the data and put it directly into the statement string.

The script shown in this recipe, *store_image.pl*, runs from the command line and stores an image file for later use. The script takes no side in the debate over whether to store images in the database or the filesystem: it implements *both* approaches! Of course, this requires twice the storage space. To adapt this script for your own use, you'll want to retain only the parts appropriate for the storage method you want to use. The necessary modifications are discussed at the end of this section.

The *store_image.pl* script uses an `image` table that includes columns for the image ID, name, and MIME type, and a column in which to store the image data:

```
CREATE TABLE image
(
  id    INT UNSIGNED NOT NULL AUTO_INCREMENT, # image ID number
  name  VARCHAR(30) NOT NULL,                 # image name
  type  VARCHAR(20) NOT NULL,                 # image MIME type
  data  LONGBLOB NOT NULL,                    # image data
  PRIMARY KEY (id),                           # id and name are unique
  UNIQUE (name)
);
```

The `name` column indicates the name of the image file in the directory where images are stored in the filesystem. The `data` column is a `LONGBLOB`, the largest `BLOB` type.

It is possible to use the `name` column to store full pathnames to images in the database, but if you put them all under the same directory, you can store names that are relative to that directory, and `name` values will take less space. That's what *store_image.pl* does. It needs to know the pathname of the image storage directory, which is what its `$image_dir` variable is for. You should check this variable's value and modify it as necessary before running the script. The default value reflects where I like to store images, but you'll need to change it according to your own preferences. Make sure to create the directory if it doesn't exist before you run the script, and set its access permissions so that the web server can read and write files there. You'll also need to check and possibly change the image directory pathname in the *display_image.pl* script discussed in Recipe 19.7.

The image storage directory should be outside the web server document tree. Otherwise, a user who knows or can guess the location may be able to upload executable code and cause it to run by requesting it with a web browser.

store_image.pl looks like this:

```
#!/usr/bin/perl
# store_image.pl: read an image file, store in the image table and
```

```
# in the filesystem.  (Normally, you'd store images only in one
# place or another; this script demonstrates how to do both.)

use strict;
use warnings;
use Fcntl;    # for O_RDONLY, O_WRONLY, O_CREAT
use FileHandle;
use Cookbook;

# Default image storage directory and pathname separator
# *** (CHANGE THESE AS NECESSARY) ***
# The location should NOT be within the web server document tree
my $image_dir = "/usr/local/lib/mcb/images";
my $path_sep = "/";

# Reset directory and pathname separator for Windows/DOS
if ($^O =~ /^MSWin/i || $^O =~ /^dos/)
{
  $image_dir = "C:\\mcb\\images";
  $path_sep = "\\";
}

-d $image_dir or die "$0: image directory ($image_dir)\ndoes not exist\n";

# Print help message if script was not invoked properly

(@ARGV == 2 || @ARGV == 3) or die <<USAGE_MESSAGE;
Usage: $0 image_file mime_type [image_name]

image_file = name of the image file to store
mime_time = the image MIME type (e.g., image/jpeg or image/png)
image_name = alternate name to give the image

image_name is optional; if not specified, the default is the
image file basename.
USAGE_MESSAGE

my $file_name = shift (@ARGV);  # image filename
my $mime_type = shift (@ARGV);  # image MIME type
my $image_name = shift (@ARGV); # image name (optional)

# if image name was not specified, use filename basename
# (permit either / or \ as separator)
($image_name = $file_name) =~ s|.*[/\\]|| unless defined ($image_name);

my $fh = new FileHandle;
my ($size, $data);

sysopen ($fh, $file_name, O_RDONLY)
  or die "Cannot read $file_name: $!\n";
binmode ($fh);    # helpful for binary data
$size = (stat ($fh))[7];
```

```
sysread ($fh, $data, $size) == $size
  or die "Failed to read entire file $file_name: $!\n";
$fh->close ();

# Save image file in filesystem under $image_dir.  (Overwrite file
# if an old version exists.)

my $image_path = $image_dir . $path_sep . $image_name;

sysopen ($fh, $image_path, O_WRONLY|O_CREAT)
  or die "Cannot open $image_path: $!\n";
binmode ($fh);    # helpful for binary data
syswrite ($fh, $data, $size) == $size
  or die "Failed to write entire image file $image_path: $!\n";
$fh->close ();

# Save image in database table.  (Use REPLACE to kick out any old image
# that has the same name.)

my $dbh = Cookbook::connect ();
$dbh->do ("REPLACE INTO image (name,type,data) VALUES(?,?,?)",
          undef,
          $image_name, $mime_type, $data);
$dbh->disconnect ();
```

If you invoke the script with no arguments, it displays a short help message. Otherwise, it requires two arguments that specify the name of the image file and the MIME type of the image. By default, the file's basename (final component) is also used as the name of the image stored in the database and in the image directory. To use a different name, provide it using an optional third argument.

The script is fairly straightforward. It implements the following procedure:

1. Check that the proper number of arguments was given and initialize some variables from them.
2. Make sure the image directory exists. If it does not, the script cannot continue.
3. Open and read the contents of the image file.
4. Store the image as a file in the image directory.
5. Store a row containing identifying information and the image data in the `image` table.

store_image.pl uses `REPLACE` rather than `INSERT` so that you can replace an old image with a new version having the same name simply by loading the new one. The statement specifies no `id` column value; `id` is an `AUTO_INCREMENT` column, so MySQL assigns a unique sequence number automatically. If you replace an image by loading a new one with the same name as an existing image, the `REPLACE` statement generates a new `id` value. To keep the old value, use `INSERT ... ON DUPLICATE KEY UPDATE` instead (see

Recipe 13.12). This inserts the row if the name doesn't already exist, or updates the image value if it does.

The `REPLACE` statement that stores the image information into MySQL is relatively mundane:

```
$dbh->do ("REPLACE INTO image (name,type,data) VALUES(?,?,?)",
          undef,
          $image_name, $mime_type, $data);
```

If you examine that statement looking for some special technique for handling binary data, you'll be disappointed, because the `$data` variable that contains the image isn't treated as special in any way. The statement refers to all column values uniformly using `?` placeholder characters and the values are passed at the end of the `do()` call. Another way to accomplish the same result is to perform escape processing on the column values explicitly, then insert them directly into the statement string:

```
$image_name = $dbh->quote ($image_name);
$mime_type = $dbh->quote ($mime_type);
$data = $dbh->quote ($data);
$dbh->do ("REPLACE INTO image (name,type,data)
           VALUES($image_name,$mime_type,$data)");
```

Image-handling has a reputation for being a lot more troublesome than it really is. If you properly handle image data in a statement by using placeholders or by encoding it, you'll have no problems. If you don't, you'll get errors. It's as simple as that. This is *no* different from how you should handle other kinds of data, even text. After all, if you insert into a statement a piece of text that contains quotes or other special characters without escaping them, the statement will blow up in your face. So the need for placeholders or encoding is not some special thing that's necessary only for images—it's necessary for all data. Say it with me: "I will always use placeholders or encode my column values. Always." (Actually, if you know enough about a given value—for example, that it's an integer—there are times when you can break this rule. Nevertheless, it's never wrong to follow the rule.)

To try the script, change location into the *apache/images* directory of the `recipes` distribution. That directory contains the *store_image.pl* script, and some sample images are in its *flags* subdirectory (they're pictures of national flags for several countries). To store one of these images, run the script like this under Unix:

```
% ./store_image.pl flags/iceland.jpg image/jpeg
```

Or like this under Windows:

```
C:\> store_image.pl flags\iceland.jpg image/jpeg
```

store_image.pl takes care of image storage, and the next section discusses how to retrieve images to serve them over the Web. What about deleting images? I'll leave it to you to write a utility to remove images that you no longer want. If you are storing images in

the filesystem, remember to delete both the database row *and* the image file the row points to.

store_image.pl stores each image both in the database and in the filesystem for illustrative purposes, but of course that makes it inefficient. Earlier, I mentioned that if you use this script as a basis for your own applications, you should modify it to store images only in one place—either in the database or in the filesystem—not in both places:

- To adapt the script to store images only in MySQL, there is no need to create an image directory, and you can delete the code that checks for that directory's existence and that writes image files there.
- To adapt the script for storage only in the filesystem, drop the `data` column from the `image` table, and modify the `REPLACE` statement to omit that column.

These modifications also apply to the *display_image.pl* image processing script shown in Recipe 19.7.

See Also

Recipe 19.7 shows how to retrieve images for display over the Web. Recipe 20.8 discusses how to upload images from a web page for storage into MySQL.

19.7. Serving Images or Other Binary Data

Problem

You can store images or other binary data values in your database, using the techniques discussed in Recipe 19.6. But how do you get them back out?

Solution

You need nothing more than a `SELECT` statement. Of course, what you *do* with the information after you retrieve it might be a little more involved.

Discussion

As described in Recipe 19.6, it's difficult to issue a statement manually that stores a literal image value into a database, so normally you use `LOAD_FILE()` or write a script that encodes the image data for insertion. However, there is no problem at all entering a statement that retrieves an image:

```
SELECT * FROM image WHERE id = 1;
```

But binary information tends not to show up well on text-display devices, so you probably don't want to do this interactively from the *mysql* program unless you want your

terminal window to turn into a horrible mess of gibberish, or possibly even lock up. It's more common to use the information for display in a web page. Or you might send it to the client for downloading (see Recipe 19.9), although that is more common for nonimage binary data such as PDF files.

To display an image in a web page, include an `<img>` tag in the page that tells the client's web browser where to get the image. If you've stored images as files in a directory to which the web server has access, you can refer to an image directly. For example, if the image file *iceland.jpg* is located in the */usr/local/lib/mcb/images* directory, refer to it like this:

```
<img src="/usr/local/lib/mcb/images/iceland.jpg"/>
```

With this approach, make sure that each image filename has an extension (such as *.gif* or *.png*) that enables the web server to determine what kind of `Content-Type:` header to generate when it sends the file to the client.

If the images are stored in a database table instead, or in a directory inaccessible to the web server, the `<img>` tag can refer to a script that knows how to fetch images and send them to clients. To do this, the script must respond by sending a `Content-Type:` header that indicates the image format, a `Content-Length:` header that indicates the number of bytes of image data, a blank line, and finally the image itself as the body of the response.

The following script, *display_image.pl*, demonstrates how to serve images over the Web. It requires a `name` parameter that indicates which image to display, and permits an optional `location` parameter that specifies whether to retrieve the image from the `image` table or from the filesystem. The default is to retrieve image data from the `image` table. For example, the following URLs display an image from the database and from the filesystem, respectively:

```
http://localhost/cgi-bin/display_image.pl?name=iceland.jpg
http://localhost/cgi-bin/display_image.pl?name=iceland.jpg;location=fs
```

The script looks like this:

```
#!/usr/bin/perl
# display_image.pl: display image over the Web

use strict;
use warnings;
use CGI qw(:standard escapeHTML);
use FileHandle;
use Cookbook;

sub error
{
my $msg = escapeHTML ($_[0]);

  print header (), start_html ("Error"), p ($msg), end_html ();
  exit (0);
```

```
}

# Default image storage directory and pathname separator
# *** (CHANGE THESE AS NECESSARY) ***
my $image_dir = "/usr/local/lib/mcb/images";
# The location should NOT be within the web server document tree
my $path_sep = "/";

# Reset directory and pathname separator for Windows/DOS
if ($^O =~ /^MSWin/i || $^O =~ /^dos/)
{
  $image_dir = "C:\\mcb\\images";
  $path_sep = "\\";
}

my $name = param ("name");
my $location = param ("location");

# make sure image name was specified
defined ($name) or error ("image name is missing");
# use default of "db" if the location is not specified or is
# not "db" or "fs"
(defined ($location) && $location eq "fs") or $location = "db";

my $dbh = Cookbook::connect ();

my ($type, $data);

# If location is "db", get image data and MIME type from image table.
# If location is "fs", get MIME type from image table and read the image
# data from the filesystem.

if ($location eq "db")
{
  ($type, $data) = $dbh->selectrow_array (
                           "SELECT type, data FROM image WHERE name = ?",
                           undef,
                           $name)
        or error ("Cannot find image with name $name");
}
else
{
  $type = $dbh->selectrow_array (
                           "SELECT type FROM image WHERE name = ?",
                           undef,
                           $name)
        or error ("Cannot find image with name $name");
  my $fh = new FileHandle;
  my $image_path = $image_dir . $path_sep . $name;
  open ($fh, $image_path)
    or error ("Cannot read $image_path: $!");
  binmode ($fh);    # helpful for binary data
```

```
  my $size = (stat ($fh))[7];
  read ($fh, $data, $size) == $size
    or error ("Failed to read entire file $image_path: $!");
  $fh->close ();
}

$dbh->disconnect ();

# Send image to client, preceded by Content-Type: and Content-Length:
# headers.

print header (-type => $type, -Content_Length => length ($data));
print $data;
```

19.8. Serving Banner Ads

Problem

You want to display banner ads by choosing images on the fly from a set of images.

Solution

Use a script that selects a random row from an image table and sends the image to the client.

Discussion

The *display_image.pl* script shown in Recipe 19.7 assumes that the URL contains a parameter that names the image to be sent to the client. Another application might determine which image to display for itself. One popular image-related use for web programming is to serve banner advertisements for display in web pages. A simple way to do this is by means of a script that picks an image at random each time it is invoked. The following Python script, *banner.py*, shows how to do this, where the "ads" are the flag images in the `image` table:

```
#!/usr/bin/python
# banner.py: serve randomly chosen banner ad from image table
# (sends no response if no image can be found)

import cookbook

conn = cookbook.connect()

stmt = "SELECT type, data FROM image ORDER BY RAND() LIMIT 1"
cursor = conn.cursor()
cursor.execute(stmt)
row = cursor.fetchone()
cursor.close()
```

```
if row is not None:
  (type, data) = row
  # Send image to client, preceded by Content-Type: and
  # Content-Length: headers.  The Expires:, Cache-Control:, and
  # Pragma: headers help keep browsers from caching the image
  # and reusing it for successive requests for this script.
  print("Content-Type: %s" % type)
  print("Content-Length: %s" % len(data))
  print("Expires: Sat, 01 Jan 2000 00:00:00 GMT")
  print("Cache-Control: no-cache")
  print("Pragma: no-cache")
  print("")
  print(data)

conn.close()
```

banner.py sends a few headers in addition to the usual `Content-Type:` and `Content-Length:` headers. The extra headers help keep browsers from caching the image. `Expires:` specifies a date in the past to tell the browser that the image is out of date. The `Cache-Control:` and `Pragma:` headers tell the browser not to cache the image. The script sends both headers because some browsers understand one, and some the other.

Why suppress caching? Because if you don't, the browser will send a request for *banner.py* only the first time it sees it in a link. On subsequent requests for the script, the browser will reuse the image, which defeats the intent of having each such link resolve to a randomly chosen image.

Install the *banner.py* script in your *cgi-bin* directory. Then, to place a banner in a web page, use an `<img>` tag that invokes the script. For example, if the script is installed as */cgi-bin/banner.py*, the following page references it to include an image below the introductory paragraph:

```
<!-- bannertest1.html: page with single link to banner-ad script -->
<html>
<head><title>Banner Ad Test Page 1</title></head>
<body>

<p>You should see an image below this paragraph.</p>

<img src="/cgi-bin/banner.py"/>

</body>
</html>
```

If you request this page, it should display an image, and you should see a succession of randomly chosen images each time you reload the page. (I am assuming here that you have loaded several images into the `image` table by now using the *store_image.pl* script discussed in Recipe 19.6. Otherwise you'll see no images at all!) If you modify *banner.py* not to send the cache-related headers, you likely will see the same image each time you reload the page.

The cache-control headers suppress caching for links to *banner.py* that occur over the course of successive page requests. Another complication occurs if multiple links to the script occur within the *same* page. The following page illustrates what happens:

```
<!-- bannertest2.html: page with multiple links to banner-ad script -->
<html>
<head><title>Banner Ad Test Page 2</title></head>
<body>

<p>You should see two images below this paragraph,
and they probably will be the same.</p>

<img src="/cgi-bin/banner.py"/>
<img src="/cgi-bin/banner.py"/>

<p>You should see two images below this paragraph,
and they probably will be different.</p>

<img src="/cgi-bin/banner.py?image1"/>
<img src="/cgi-bin/banner.py?image2"/>

</body>
</html>
```

The first pair of links to *banner.py* are identical. What you'll probably find when you request this page is that your browser notices that fact, sends only a single request to the web server, and uses the image that is returned where both links appear in the page. As a result, the first pair of images displayed in the page will be identical. The second pair of links to *banner.py* show how to solve this problem. The links include some extra fluff at the end of the URLs that make them look different. *banner.py* doesn't use that information at all, but making the links look different fools the browser into sending two image requests. The result is that the second pair of images differ from each other, unless *banner.py* happens to randomly select the same image both times.

19.9. Serving Query Results for Download

Problem

You want to send database information to a browser for downloading rather than for display.

Solution

Unfortunately, there's no good way to force a download. A browser processes information sent to it according to the `Content-Type:` header value, and if it has a handler for that value, it treats the information accordingly. However, you may be able to trick the browser by using a "generic" content type for which it's unlikely to have a handler.

Discussion

Earlier sections of this chapter discuss how to incorporate the results of database queries into web pages, to display them as paragraphs, lists, tables, or images. But what if you want to produce a query result that the user can download to a file instead? It's not difficult to generate the response itself: send a `Content-Type:` header preceding the information, such as `text/plain` for plain text, `image/jpeg` for a JPEG image, or `application/pdf` or `application/msexcel` for a PDF or Excel document. Then send a blank line and the content of the query result. The problem is that there's no way to force the browser to download the information. If it knows what to do with the response based on the content type, it will try to handle the information as it sees fit. If it knows how to display text or images, it will. If it thinks it's supposed to give a PDF or Excel document to a PDF viewer or to Excel, it will. Most browsers enable the user to select a download explicitly (for example, by right-clicking or Ctrl-clicking a link), but that's a client-side mechanism. You have no access to it on the server end.

What you can do is fool the browser by faking the content type. The most generic type is `application/octet-stream`. Most users are unlikely to have any content handler specified for it, so if you send a response using that type, it's likely to trigger a download by the browser. The disadvantage of this, of course, is that the response contains a false indicator about the type of information it contains. You can try to alleviate this problem by suggesting a default filename for the browser to use when it saves the file. If the filename has a suffix indicative of the file type, such as *.txt*, *.jpg*, *.pdf*, or *.xls*, that may help the client (or the operating system on the client host) determine how to process the file. To suggest a name, include a `Content-Disposition:` header in the response:

```
Content-disposition: attachment; filename="suggested_name"
```

The following PHP script, *download.php*, demonstrates one way to produce downloadable content. When first invoked, it presents a page containing a link that can be selected to initiate the download. The link points back to *download.php* but includes a `download` parameter. When you select the link, it reinvokes the script, which sees the parameter and responds by issuing a query, retrieving a result set, and sending it to the browser for downloading. The `header()` function sets the `Content-Type:` and `Content-Disposition:` headers in the response. (Do this before the script produces any other output, or `header()` has no effect.)

```
<?php
# download.php: retrieve result set and send it to user as a download
# rather than for display in a web page

require_once "Cookbook.php";
require_once "Cookbook_Webutils.php";

$title = "Result Set Downloading Example";

# If no download parameter is present, display instruction page
```

```
if (!get_param_val ("download"))
{
  # construct self-referential URL that includes download parameter
  $url = $_SERVER["PHP_SELF"] . "?download=1";
?>

<html>
<head><title><?php print ($title); ?></title></head>
<body>
<p>
Select the following link to commence downloading:
<a href="<?php print ($url); ?>">download</a>
</p>
</body>
</html>

<?php
  exit ();
} # end of "if"

# The download parameter was present; retrieve a result set and send
# it to the client as a tab-delimited, newline-terminated document.
# Use a content type of application/octet-stream in an attempt to
# trigger a download response by the browser, and suggest a default
# filename of "result.txt".

$dbh = Cookbook::connect ();

$stmt = "SELECT * FROM profile";
$sth = $dbh->query ($stmt);

header ('Content-Type: application/octet-stream');
header ('Content-Disposition: attachment; filename="result.txt"');

while ($row = $sth->fetch (PDO::FETCH_NUM))
  print (implode ("\t", $row) . "\n");

$dbh = NULL;
?>
```

download.php uses a `get_param_val()` function that we haven't covered yet. It determines whether that parameter is present. This function is included in the *Cookbook_Webutils.php* file and discussed further in Recipe 20.5.

Another way to produce downloadable content is to generate the query result, write it to a file on the server side, compress it, and send the result to the browser. The browser likely will download it and run some kind of uncompress utility to recover the original file.

CHAPTER 20

Processing Web Input with MySQL

20.0. Introduction

The previous chapter describes how to retrieve information from MySQL and display it in web pages using HTML constructs such as tables or hyperlinks. That's a use of MySQL to send information in one direction (from web server to user). This chapter considers the use of MySQL in the other direction: web-based database programming to collect information sent from user to web server, such as the contents of a submitted form. For example, you might store the information from a survey form for later use, or use keywords from a lookup form as the basis for a query to search the database for information the user wants.

MySQL comes into these activities as the repository for storing information or as the source from which search results are drawn. But before you can process input from a form, you must create the form and send it to the user. MySQL helps with this, too, because it's often possible to use information from your database to generate form elements such as radio buttons, checkboxes, pop-up menus, or scrolling lists:

- Select a set of items from a table that lists countries, states, or provinces and convert them into a pop-up menu in a form that collects address information.
- Use the list of legal values for an `ENUM` column that contains permitted colors or sizes to generate a set of radio buttons.
- Use lists of available colors, sizes, or styles stored in an inventory database to construct fields for a clothing ordering form.

Using database content to generate form elements lessens the amount of explicit knowledge your programs must have about table structure and content, and enables them to determine what they need automatically. A script that uses database content to generate form elements also adaptively handles changes to the database. To add a new country, create a new row in the table that stores the list of countries. To add a new salutation,

change the definition of the ENUM column. In each case, you change the set of items in a form element by updating the database, not by modifying the script; the script adapts to the change without additional programming.

The first part of this chapter covers the following topics relating to web input processing:

Generating forms and form elements
: One way to use database content for form construction is to select a list of items from a table and create the options in a list element. But metadata can be used as well. There is a natural correspondence between ENUM columns and single-pick form elements like radio button sets or pop-up menus. In both cases, only one from a set of possible values may be chosen. There is a similar correspondence between SET columns and multiple-pick elements like checkbox groups; any or all of the possible values may be chosen.

Initializing forms using database contents
: In addition to using the database to create structural elements of forms, you can also use it to initialize their values. For example, to enable a user to modify an existing record, retrieve it from the database and load its values into the corresponding form fields before sending the form to the user for editing.

Processing input gathered over the Web
: This includes input not only from form fields, but also the contents of uploaded files, or parameters in URLs. Regardless of how you obtain the information, you face a common set of issues in dealing with it: extracting and decoding the information, performing constraint or validity checking on it, and re-encoding the information for SQL statement construction to avoid generating malformed statements or storing information inaccurately.

The second part of the chapter illustrates how to apply the techniques developed in the first part. These include applications that show how to use MySQL to present a web-based search interface, create paged displays that contain next-page and previous-page links, implement per-page hit counting and logging, and perform Apache logging to a database.

Scripts to create tables used in this chapter are located in the *tables* directory of the recipes distribution. Scripts for the examples are located under the directories named for the web servers used to run them. For Perl, Ruby, PHP, and Python examples, look under the *apache* directory. Utility routines used by the example scripts are found in files located in the *lib* directory. (For information on configuring Apache so that scripts run by it can find their library files, see Recipe 18.2.) For Java (JSP) examples, look under the *tomcat* directory; you should already have installed these in the process of setting up the mcb application context (see Recipe 18.3).

If a particular section has no example for a language in which you're interested, check the recipes distribution for implementations not shown here.

The scripts in this chapter are intended to be invoked from your browser after they have been installed, but you can invoke many of them (JSP pages excepted) from the command line to see the raw HTML they produce; see Recipe 18.2.

To provide a concrete context for discussion, many of the form-processing examples in this chapter are based on the following scenario. You run a business in the lucrative "construct-a-cow" market that manufactures built-to-order ceramic bovine figurines, and you want to design an online ordering application that lets customers make selections for several aspects of the product. For each order, it's necessary to collect several types of information:

Cow color
: The particular list of colors available at any particular time changes occasionally, so for flexibility, the values can be stored in a database table. To change the set of colors from which customers can choose, just update the table.

Cow size
: There is a fixed set of sizes that doesn't change often (small, medium, large), so the values can be represented as elements of an `ENUM` column.

The all-important cow accessory items
: These include a bell, horns, a sporty-looking tail ribbon, and a nose ring. Accessories can be represented in a `SET` column because a customer may want to select more than one of them. In addition, you know from past experience that most customers order horns and a cow bell, so it's reasonable to use those for the column's default value.

Customer name and address (street, city, state)
: The possible state names are already stored in the `states` table. We can use them as the basis for the corresponding form element.

Given the preceding discussion, a `cow_order` table can be designed like this:

```
CREATE TABLE cow_order
(
  id          INT UNSIGNED NOT NULL AUTO_INCREMENT,
  # cow color, figurine size, and accessory items
  color       VARCHAR(20),
  size        ENUM('small','medium','large') DEFAULT 'medium',
  accessories SET('cow bell','horns','nose ring','tail ribbon')
                  DEFAULT 'cow bell,horns',
  # customer name, street, city, and state (abbreviation)
  cust_name   VARCHAR(40),
  cust_street VARCHAR(40),
  cust_city   VARCHAR(40),
  cust_state  CHAR(2),
  PRIMARY KEY (id)
);
```

The `id` column provides a unique identifier for each row. This value is needed when we get to Recipe 20.4, which shows how to use web forms to edit existing rows. That task requires being able to tell which row to update, which is difficult to do without a unique identifier.

For the list of available colors, we maintain a separate table, `cow_color`:

```
CREATE TABLE cow_color (color CHAR(20));
```

Assume for purposes of illustration that the `cow_color` table contains the following rows:

```
+---------------+
| color         |
+---------------+
| Black         |
| Black & White |
| Brown         |
| Cream         |
| Red           |
| Red & White   |
| See-Through   |
+---------------+
```

An application can use the tables just described to generate list elements in an order entry form. The next several recipes describe how to do this, and how to process the input that you obtain when a user submits a form, without requiring the application to have specialized built-in knowledge about the available options.

20.1. Writing Scripts That Generate Web Forms

Problem

You want to write a script that gathers input from a user.

Solution

Create a form within your script and send it to the user. The script can arrange to have itself invoked again to process the form's contents when the user fills it in and submits it.

Discussion

Web forms are a convenient way to enable your visitors to submit information such as a set of search keywords, a completed survey result, or a response to a questionnaire. Forms are also beneficial for you as a developer because they provide a structured way to associate data values with names by which to refer to them.

A form begins and ends with `<form>` and `</form>` tags. Place other HTML constructs between those tags, including elements that become input fields in the page that the browser displays. The `<form>` tag that begins a form should include two attributes, `action` and `method`. The `action` attribute tells the browser what to do with the form when the user submits it. This is the URL of the script to invoke to process the form's contents. The `method` attribute indicates to the browser what kind of HTTP request to use to submit the form. The value is either `get` or `post`. Recipe 20.5 discusses the difference between these two request methods; for now, we'll always use `post`.

Most of the form-based web scripts shown in this chapter share some common behaviors:

- When first invoked, the script generates a form and sends it to the user to be filled in.
- The `action` attribute of the form points back to the same script. When the user completes the form and submits it, the web server invokes the script again to process the form's contents.
- The script checks its execution environment to see what input parameters are present. For the initial invocation, the environment contains none of the parameters named in the form. This enables the script to determine whether it's being invoked by a user for the first time or whether it should process a submitted form.

This approach isn't the only one you can adopt. One alternative is to place a form in a static HTML page and have it point to the script that processes the form. Another is to have one script generate the form and a second script process it.

If a form-creating script wants to have itself invoked again when the user submits the form, it should determine its own pathname within the web server document tree and use that value for the `action` attribute of the opening `<form>` tag. For example, if a script is installed as */cgi-bin/myscript* in your web tree, you could write the `<form>` tag like this:

```
<form action="/cgi-bin/myscript" method="post">
```

Each of our language APIs provides a way for a script to obtain its own pathname, which enables you to avoid hardwiring a script's pathname into it and gives you greater latitude where to install it.

Perl

In Perl scripts, the CGI.pm module provides three useful methods for creating `<form>` elements and constructing the `action` attribute. `start_form()` and `end_form()` generate the opening and closing form tags, and `url()` returns the script's own pathname. Using these methods, scripts generate a form like this:

```
print start_form (-action => url (), -method => "post");
# ... generate form elements here ...
print end_form ();
```

`start_form()` supplies a default request method of `post`. You can omit the `method` argument if you're constructing a `post` form.

Ruby

In Ruby scripts, create a `cgi` object, and use its `form` method to generate a form. The method arguments provide the `<form>` tag attributes, and the block following the method call provides the form content. To get the script pathname, use the `SCRIPT_NAME` member of the `ENV` hash:

```
cgi.out {
  cgi.form("action" => ENV["SCRIPT_NAME"], "method" => "post") {
    # ... generate form elements here ...
  }
}
```

The `form` method supplies a default request method of `post`. You can omit the `method` argument if you're constructing a `post` form.

The script pathname is also available from the `cgi.script_name` method.

PHP

PHP scripts access the script pathname as the `PHP_SELF` member of the `$_SERVER` array, which is a "superglobal" array (accessible in any scope without being declared as global). Scripts can obtain their own pathname and use it to generate a form as follows:

```
print ('<form action="' . $_SERVER['PHP_SELF'] . '" method="post">');
# ... generate form elements here ...
print ('</form>');
```

Python

Python scripts get the script pathname by importing the `os` module and accessing the `SCRIPT_NAME` member of the `os.environ` object:

```
import os

print('<form action="%s" method="post">' % os.environ['SCRIPT_NAME'])
# ... generate form elements here ...
print('</form>')
```

Java

In JSP pages, the request path is available through the implicit `request` object provided by the JSP processor. Use that object's `getRequestURI()` method as follows:

```
<form action="<%= request.getRequestURI () %>" method="post">
<%-- ... generate form elements here ... --%>
</form>
```

See Also

The examples shown in this section have an empty body between the opening and closing form tags. For a form to be useful, you must create body elements that correspond to the types of information to be obtained from users. It's possible to hardwire these elements into a script, but Recipes 20.2 and 20.3 describe how MySQL helps you create the elements on the fly from information stored in your database.

20.2. Creating Single-Pick Form Elements from Database Content

Problem

A form must present a field that enables the user to select one of several options.

Solution

Use a single-pick list element. These include radio button sets, pop-up menus, and scrolling lists.

Discussion

Single-pick form elements enable you to present multiple choices from which a single option can be selected. Our construct-a-cow scenario (see Recipe 20.0) involves several sets of single-pick choices:

- The list of colors in the `cow_color` table. These can be obtained with the following statement:

```
mysql> SELECT color FROM cow_color ORDER BY color;
+---------------+
| color         |
+---------------+
| Black         |
| Black & White |
| Brown         |
| Cream         |
| Red           |
| Red & White   |
| See-Through   |
+---------------+
```

Some of the colors contain an & character, which is special in HTML and must be HTML-encoded when placed into list elements. (We'll perform list element encoding as a matter of habit. Those values illustrate why it's a good idea to get in that habit.)

- The list of legal figurine sizes in the `size` column of the `cow_order` table. The column is represented as an `ENUM`, so the permitted and default values can be obtained from `INFORMATION_SCHEMA`:

```
mysql> SELECT COLUMN_TYPE, COLUMN_DEFAULT
    -> FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA='cookbook' AND TABLE_NAME='cow_order'
    -> AND COLUMN_NAME='size';
+--------------------------------+----------------+
| COLUMN_TYPE                    | COLUMN_DEFAULT |
+--------------------------------+----------------+
| enum('small','medium','large') | medium         |
+--------------------------------+----------------+
```

- The list of state names and abbreviations. These are stored in the `states` table:

```
mysql> SELECT abbrev, name FROM states ORDER BY name;
+--------+----------------+
| abbrev | name           |
+--------+----------------+
| AL     | Alabama        |
| AK     | Alaska         |
| AZ     | Arizona        |
| AR     | Arkansas       |
| CA     | California     |
| CO     | Colorado       |
…
```

The number of choices varies for each list just described: 3 figurine sizes, 7 colors, and 50 states. The differing numbers of choices lead to different decisions about how to represent the lists in a form:

- The figurine size values are best represented as a set of radio buttons or a pop-up menu; a scrolling list is unnecessary because the number of choices is small.
- The set of colors can reasonably be displayed using any of the single-pick element types; it's small enough that a set of radio buttons wouldn't take a lot of space, but large enough that you may want to enable scrolling—particularly if you make additional colors available.
- The list of states has more items than feasible to present as a set of radio buttons. It's more reasonably displayed as a pop-up menu or scrolling list.

The following discussion describes the HTML syntax for these types of elements and then shows how to generate them from within scripts:

Radio buttons

A group of radio buttons consists of `<input>` elements of type `radio`, all with the same `name` attribute. Each element also includes a `value` attribute. A label to display can be given after the `<input>` tag. To mark an item as the default initial selection, add a `checked` attribute. The following radio button group displays the possible cow figurine sizes, using `checked` to mark `medium` as the initially selected value:

```
<input type="radio" name="size" value="small" />small
<input type="radio" name="size" value="medium" checked="checked" />medium
<input type="radio" name="size" value="large" />large
```

Pop-up menus

A pop-up menu begins and ends with `<select>` and `</select>` tags, with each item in the menu enclosed within `<option>` and `</option>` tags. Each `<option>` element has a `value` attribute, and its body provides a label to be displayed. To indicate a default selection, add a `selected` attribute to the appropriate `<option>` item. If no item is so marked, the first item becomes the default, as is the case for the following pop-up menu:

```
<select name="color">
<option value="Black">Black</option>
<option value="Black &amp;amp; White">Black &amp;amp; White</option>
<option value="Brown">Brown</option>
<option value="Cream">Cream</option>
<option value="Red">Red</option>
<option value="Red &amp;amp; White">Red &amp;amp; White</option>
<option value="See-Through">See-Through</option>
</select>
```

Scrolling lists

A scrolling list displays as a set of items in a box. The list may contain more items than are visible in the box, in which case the browser displays a scrollbar so the user can bring the other items into view. The HTML syntax for scrolling lists is similar to that for pop-up menus, except that the opening `<select>` tag includes a `size` attribute indicating how many rows of the list should be visible in the box. By default, a scrolling list is a single-pick element; Recipe 20.3 discusses how to enable multiple picks.

The following single-pick scrolling list includes an item for each US state, of which six at a time are visible:

```
<select name="state" size="6">
<option value="AL">Alabama</option>
<option value="AK">Alaska</option>
<option value="AZ">Arizona</option>
<option value="AR">Arkansas</option>
<option value="CA">California</option>
…
<option value="WV">West Virginia</option>
```

```
<option value="WI">Wisconsin</option>
<option value="WY">Wyoming</option>
</select>
```

Radio button sets, pop-up menus, and scrolling lists have several things in common:

A name for the element
: When the user submits the form, the browser associates this name with the value the user selected.

A set of values, one for each item in the list
: The internal values available to be selected.

A set of labels, one for each item
: The values that the user sees in the displayed form.

An optional default value
: Which item in the list is selected initially when the browser displays the list.

To produce a list element for a form using database content, issue a statement that selects the appropriate values and labels, encode any special characters they contain, and add the HTML tags that are appropriate for the kind of list you want to display. Should you desire to indicate a default selection, add a `checked` or `selected` attribute to the proper item in the list.

Let's consider how to produce form elements for the color and state lists first. Both are produced by fetching a set of column values from a table. Later we'll construct the figurine size list, which takes its values from a column's definition (that is, its metadata) rather than its contents.

In JSP, display a set of radio buttons for the colors using JSTL tags as follows. The color names are used as the values as well as the labels, so they're printed twice:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT color FROM cow_color ORDER BY color
</sql:query>

<c:forEach items="${rs.rows}" var="row">
  <input type="radio" name="color"
    value="<c:out value="${row.color}"/>"
  /><c:out value="${row.color}"/><br />
</c:forEach>
```

`<c:out>` performs HTML entity encoding, so the `&` character that is present in some of the color values is converted to `&amp;` automatically and causes no display problems in the resulting web page. (For JSTL background, read "JSP, JSTL, and Tomcat Primer" on the companion website; see the Preface).

To display a pop-up menu instead, the retrieval statement is the same, but the row-fetching loop differs:

```
<select name="color">
<c:forEach items="${rs.rows}" var="row">
  <option value="<c:out value="${row.color}"/>">
  <c:out value="${row.color}"/></option>
</c:forEach>
</select>
```

The pop-up menu can be changed easily to a scrolling list; add a `size` attribute to the opening `<select>` tag. For example, to make three colors visible at a time, generate the list like this:

```
<select name="color" size="3">
<c:forEach items="${rs.rows}" var="row">
  <option value="<c:out value="${row.color}"/>">
  <c:out value="${row.color}"/></option>
</c:forEach>
</select>
```

Generating a list element for the set of states is similar, except that the labels are not the same as the values. To make the labels more meaningful to customers, display the full state names. But the value returned when the form is submitted should be an abbreviation because that is what gets stored in the `cow_order` table. To produce a list that way, select both the abbreviations and the full names and insert them into the proper parts of each list item. For example, to create a pop-up menu, do this:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT abbrev, name FROM states ORDER BY name
</sql:query>

<select name="state">
<c:forEach items="${rs.rows}" var="row">
  <option value="<c:out value="${row.abbrev}"/>">
  <c:out value="${row.name}"/></option>
</c:forEach>
</select>
```

The preceding JSP examples use an approach that prints each list item individually. List element generation in CGI.pm-based Perl scripts proceeds on a different basis: extract the information from the database first, and then pass it all to a function that returns a string representing the form element. The functions that generate single-pick elements are `radio_group()`, `popup_menu()`, and `scrolling_list()`. These have several arguments in common:

`name`
: The list element name.

`values`
: The values for the items in the list. This should be a reference to an array.

`labels`
: The labels to associate with each value. This argument is optional; if it's missing, CGI.pm uses the values as the labels. Otherwise, the `labels` argument should be a reference to a hash that associates each value with its corresponding label. For example, to produce a list element for cow colors, the values and labels are the same, so no `labels` argument is necessary. However, to produce a state list, `labels` should be a reference to a hash that maps each state abbreviation to its full name.

`default`
: The initially selected item in the element. This argument is optional. For a radio button set, CGI.pm automatically selects the first button by default if this argument is missing. To defeat that behavior, provide a default value not present in the `values` list. (This value cannot be `undef` or the empty string.)

Some of the functions take additional arguments. For `radio_group()`, you can supply a `linebreak` argument to specify that the buttons should be displayed vertically rather than horizontally. `scrolling_list()` takes a `size` argument indicating how many items should be visible at a time. (The CGI.pm documentation describes additional arguments that are not used here at all. For example, there are arguments for laying out radio buttons in tabular form. We won't be that fancy.)

To construct a form element using the colors in the `cow_color` table, begin by retrieving them as an array:

```
my $color_ref = $dbh->selectcol_arrayref (qq{
  SELECT color FROM cow_color ORDER BY color
});
```

`selectcol_arrayref()` returns a reference to the array, exactly the kind of value needed for the `values` argument of the CGI.pm functions that create list elements. To create a group of radio buttons, a pop-up menu, or a single-pick scrolling list, invoke the functions as follows:

```
print radio_group (-name      => "color",
                   -values    => $color_ref,
                   -linebreak => 1);      # display buttons vertically

print popup_menu (-name   => "color",
                  -values => $color_ref);

print scrolling_list (-name   => "color",
                      -values => $color_ref,
                      -size   => 3);      # display 3 items at a time
```

The values and the labels for the color list are the same, so no `labels` argument need be given; CGI.pm uses the values as labels by default. Note that we haven't HTML-encoded the colors here, even though some of them contain an `&` character. CGI.pm

functions that generate form elements automatically perform HTML-encoding, unlike its functions that create nonform elements.

To produce a list of states for which the values are abbreviations and the labels are full names, we do need a `labels` argument. It should be a reference to a hash that maps each value to the corresponding label. Construct the value list and label hash as follows:

```
my @state_values;
my %state_labels;
my $sth = $dbh->prepare (qq{
  SELECT abbrev, name FROM states ORDER BY name
});
$sth->execute ();
while (my ($abbrev, $name) = $sth->fetchrow_array ())
{
  push (@state_values, $abbrev);  # save each value in an array
  $state_labels{$abbrev} = $name; # map each value to its label
}
```

Pass the resulting list and hash by reference to `popup_menu()` or `scrolling_list()`, depending on the kind of list element to produce:

```
print popup_menu (-name   => "state",
                  -values => \@state_values,
                  -labels => \%state_labels);

print scrolling_list (-name   => "state",
                      -values => \@state_values,
                      -labels => \%state_labels,
                      -size   => 6);      # display 6 items at a time
```

Like CGI.pm, the Ruby `cgi` module has methods for generating radio buttons, pop-up menus, and scrolling lists. Examine the *form_element.rb* script to see how to use them. However, I don't discuss them here because I find them awkward to use, particularly when it's necessary to ensure that values are properly escaped or that certain group members are selected by default.

If you use an API that provides no ready-made set of functions for producing form elements (or which, like Ruby `cgi`, is inconvenient to use), you may elect either to print HTML as you fetch list items from MySQL, or write utility routines that generate the form elements for you. The following discussion considers how to implement both approaches, using PHP and Python.

In PHP, to present the list of values from the `cow_color` table in a pop-up menu, use a fetch-and-print loop like this:

```
$stmt = "SELECT color FROM cow_color ORDER BY color";
$sth = $dbh->query ($stmt);
print ('<select name="color">');
while (list ($color) = $sth->fetch (PDO::FETCH_NUM))
{
```

```
  $color = htmlspecialchars ($color);
  printf ('<option value="%s">%s</option>', $color, $color);
}
print ("</select>\n");
```

Python code to do the same is similar:

```
stmt = "SELECT color FROM cow_color ORDER BY color"
cursor = conn.cursor()
cursor.execute(stmt)
print('<select name="color">')
for (color, ) in cursor:
  color = cgi.escape(color, 1)
  print('<option value="%s">%s</option>' % (color, color))
print('</select>')
cursor.close()
```

The state list requires different values and labels, so the code is slightly more complex. In PHP, it looks like this:

```
$stmt = "SELECT abbrev, name FROM states ORDER BY name";
$sth = $dbh->query ($stmt);
print ('<select name="state">');
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  $abbrev = htmlspecialchars ($row[0]);
  $name = htmlspecialchars ($row[1]);
  printf ('<option value="%s">%s</option>', $abbrev, $name);
}
print ("</select>");
```

And in Python, like this:

```
stmt = "SELECT abbrev, name FROM states ORDER BY name"
cursor = conn.cursor()
cursor.execute(stmt)
print('<select name="state">')
for (abbrev, name) in cursor:
  abbrev = cgi.escape(abbrev, 1)
  name = cgi.escape(name, 1)
  print('<option value="%s">%s</option>' % (abbrev, name))
print('</select>')
cursor.close()
```

Radio buttons and scrolling lists can be produced in similar fashion. But rather than doing so, let's use a different approach and construct a set of functions that generate form elements, given the proper information. The functions return a string representing the appropriate kind of form element. Invoke them as follows:

```
make_radio_group (name, values, labels, default, vertical)
make_popup_menu (name, values, labels, default)
make_scrolling_list (name, values, labels, default, size, multiple)
```

These functions have several arguments in common:

`name`
: The form element name.

`values`
: An array or list of values for the items in the element.

`labels`
: Another array that provides the corresponding element label to display for each value. The two arrays must be the same size. (To use the values as the labels, pass the same array to the function twice.)

`default`
: The initial value of the form element. This should be a scalar value, except for `make_scrolling_list()`. We'll write that function to handle either single-pick or multiple-pick lists (and use it for the latter purpose in Recipe 20.3), so its `default` value is permitted to be either a scalar or an array. If there is no default, pass a value not present in the `values` array; typically, an empty string will do.

Some of the functions have additional arguments that apply only to particular element types:

`vertical`
: This applies to radio button groups. If true, items are stacked vertically rather than horizontally.

`size`, `multiple`
: These arguments apply to scrolling lists. `size` indicates how many items in the list are visible, and `multiple` should be true if the list permits multiple selections.

The implementation of some of these list-generating functions is discussed here, but you can find the code for all of them in the *lib* directory of the `recipes` distribution. All of them act like CGI.pm for form element functions in the sense that they automatically perform HTML-encoding of argument values that are incorporated into the list. (The Ruby version of the library file includes utility methods for generating these elements, too, even though the `cgi` module has methods for creating them. I think the utility methods are easier to use than the `cgi` methods.)

In PHP, the `make_radio_group()` function for creating a set of radio buttons looks like this:

```
function make_radio_group ($name, $values, $labels, $default, $vertical)
{
  $result = '';
  for ($i = 0; $i < count ($values); $i++)
  {
    # select the item if it corresponds to the default value
    $checked = ($values[$i] == $default ? ' checked="checked"' : '');
    $result .= sprintf (
                '<input type="radio" name="%s" value="%s"%s />%s',
```

```
                htmlspecialchars ($name),
                htmlspecialchars ($values[$i]),
                $checked,
                htmlspecialchars ($labels[$i]));
    if ($vertical)
      $result .= '<br />'; # display items vertically
  }
  return ($result);
}
```

The function constructs the form element as a string, which it returns. To use `make_radio_group()` to present cow colors, invoke it after fetching the items from the `cow_color` table, as follows:

```
$stmt = "SELECT color FROM cow_color ORDER BY color";
$sth = $dbh->query ($stmt);
$values = $sth->fetchAll (PDO::FETCH_COLUMN, 0);
print (make_radio_group ("color", $values, $values, "", TRUE));
```

Pass the `$values` array to `make_radio_group()` twice because it's used both for the values and the labels.

To present a pop-up menu, use the following function instead:

```
function make_popup_menu ($name, $values, $labels, $default)
{
  $result = '';
  for ($i = 0; $i < count ($values); $i++)
  {
    # select the item if it corresponds to the default value
    $checked = ($values[$i] == $default ? ' selected="selected"' : '');
    $result .= sprintf (
                '<option value="%s"%s>%s</option>',
                htmlspecialchars ($values[$i]),
                $checked,
                htmlspecialchars ($labels[$i]));
  }
  $result = sprintf (
              '<select name="%s">%s</select>',
              htmlspecialchars ($name),
              $result);
  return ($result);
}
```

`make_popup_menu()` has no `$vertical` parameter, but otherwise invoke it the same way as `make_radio_group()`:

```
print (make_popup_menu ("color", $values, $values, ""));
```

The `make_scrolling_list()` function is similar to `make_popup_menu()`, so I don't show its implementation here. To invoke it to produce a single-pick list, pass the same arguments as for `make_popup_menu()`, but indicate how many rows should be visible at once, and add a `multiple` argument of `FALSE`:

```
print (make_scrolling_list ("color", $values, $values, "", 3, FALSE));
```

The state list uses labels that differ from the values. Fetch the labels and values like this:

```
$values = array ();
$labels = array ();
$stmt = "SELECT abbrev, name FROM states ORDER BY name";
$sth = $dbh->query ($stmt);
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  $values[] = $row[0];
  $labels[] = $row[1];
}
```

Use the values and labels to generate the type of list you want:

```
print (make_popup_menu ("state", $values, $labels, ""));

print (make_scrolling_list ("state", $values, $labels, "", 6, FALSE));
```

Ruby and Python implementations of the utility functions are similar to the PHP versions. For example, the Python version of `make_popup_menu()` looks like this:

```
def make_popup_menu(name, values, labels, default):
  result = ''
  # make sure name and default are strings
  name = str(name)
  default = str(default)
  for i in range(len(values)):
    # make sure value and label are strings
    value = str(values[i])
    label = str(labels[i])
    # select the item if it corresponds to the default value
    if value == default:
      checked = ' selected="selected"'
    else:
      checked = ''
    result += '<option value="%s"%s>%s</option>' % (
                cgi.escape(value, 1),
                checked,
                cgi.escape(label, 1))

  result = '<select name="%s">%s</select>' % (
                 cgi.escape(name, 1), result)
  return result
```

To present the cow colors in a form, fetch them like this:

```
values = []
stmt = "SELECT color FROM cow_color ORDER BY color"
cursor = conn.cursor()
cursor.execute(stmt)
for (color, ) in cursor:
```

```
    values.append(color)
cursor.close()
```

Then convert the list to a form element using one of the following calls:

```
print(make_radio_group('color', values, values, '', True))

print(make_popup_menu('color', values, values, ''))

print(make_scrolling_list('color', values, values, '', 3, False))
```

To present the state list, fetch the names and abbreviations:

```
values = []
labels = []
stmt = "SELECT abbrev, name FROM states ORDER BY name"
cursor = conn.cursor()
cursor.execute(stmt)
for (abbrev, name) in cursor:
  values.append(abbrev)
  labels.append(name)
cursor.close()
```

Then pass them to the appropriate function:

```
print(make_popup_menu('state', values, labels, ''))

print(make_scrolling_list('state', values, labels, '', 6, False))
```

The Ruby and Python utility methods in the *lib* directory do something that their PHP counterparts do not: explicitly convert to string form all argument values that get incorporated into the list. (You can see this in the Python version of `make_popup_menu()` earlier.) This conversion is necessary because the Ruby `CGI.escapeHTML()` and Python `cgi.escape()` methods raise an exception if you pass nonstring values to them.

We have thus far considered how to fetch rows from the `cow_color` and `states` tables and convert them to form elements. Another element in the form for the online cow-ordering application is the field for specifying cow figurine size. The legal values for this field come from the definition of the `size` column in the `cow_order` table. That column is an `ENUM`, so getting the legal values for the corresponding form element is a matter of getting the column definition and parsing it. In other words, use the column metadata rather than the column data.

As it happens, most of the work for this task has already been done in Recipe 10.7, which develops utility routines to get `ENUM` or `SET` column metadata. In Perl, for example, invoke the `get_enumorset_info()` function as follows to get the `size` column metadata:

```
my $size_info = get_enumorset_info ($dbh, "cookbook", "cow_order", "size");
```

The resulting `$size_info` value is a reference to a hash that has several members, two of which are relevant to our purposes here:

```
$size_info->{values}
$size_info->{default}
```

The `values` member is a reference to a list of the legal enumeration values, and `default` is the column's default value. Convert this information to a form element such as a group of radio buttons or a pop-up menu as follows:

```
print radio_group (-name      => "size",
                   -values    => $size_info->{values},
                   -default   => $size_info->{default},
                   -linebreak => 1);   # display buttons vertically

print popup_menu (-name    => "size",
                  -values  => $size_info->{values},
                  -default => $size_info->{default});
```

The default value is `medium`, so the browser selects that value initially when it displays the form.

The equivalent Ruby metadata-fetching method returns a hash. Use it as follows to generate form elements from the `size` column metadata:

```
size_info = get_enumorset_info(dbh, "cookbook", "cow_order", "size")

form << make_radio_group("size",
                         size_info["values"],
                         size_info["values"],
                         size_info["default"],
                         true)    # display items vertically

form << make_popup_menu("size",
                        size_info["values"],
                        size_info["values"],
                        size_info["default"])
```

The metadata function for PHP returns an associative array, which is used in similar fashion:

```
$size_info = get_enumorset_info ($dbh, "cookbook", "cow_order", "size");

print (make_radio_group ("size",
                         $size_info["values"],
                         $size_info["values"],
                         $size_info["default"],
                         TRUE));   # display items vertically

print (make_popup_menu ("size",
                        $size_info["values"],
                        $size_info["values"],
                        $size_info["default"]));
```

The Python version of the metadata function returns a dictionary:

```
size_info = get_enumorset_info(conn, 'cookbook', 'cow_order', 'size')

print(make_radio_group('size',
                       size_info['values'],
                       size_info['values'],
                       size_info['default'],
                       True))  # display items vertically

print(make_popup_menu('size',
                      size_info['values'],
                      size_info['values'],
                      size_info['default']))
```

When you use `ENUM` values like this to create list elements, values are displayed in the order they are listed in the column definition. To produce a different display order, sort the values appropriately.

To demonstrate how to process column metadata to generate form elements in JSP pages, I'll use a function embedded into the page. A better approach would be to write a custom action in a tag library that maps onto a class that returns the information, but custom tag writing is beyond the scope of this book. The examples take the following approach instead:

1. Use JSTL tags to query `INFORMATION_SCHEMA` for the `ENUM` column definition and move the definition into page context.
2. Invoke a function that extracts the definition from page context, parses it into an array of individual enumeration values, and moves the array back into page context.
3. Access the array using a JSTL iterator that displays each of its values as a list item. For each value, compare it to the column's default value and mark it as the initially selected item if it's the same.

The function that extracts legal values from an `ENUM` or `SET` column definition is named `getEnumOrSetValues()`. Place it into a JSP page like this:

```
<%@ page import="java.util.*" %>
<%@ page import="java.util.regex.*" %>

<%!
// declare a class method for breaking apart ENUM/SET values.
// typeDefAttr - the name of the page context attribute that contains
// the column type definition
// valListAttr - the name of the page context attribute in which to
// store the column value list

void getEnumOrSetValues (PageContext ctx,
                         String typeDefAttr,
                         String valListAttr)
{
  String typeDef = ctx.getAttribute (typeDefAttr).toString ();
```

```
    List values = new ArrayList ();

    // column must be an ENUM or SET
    Pattern pc = Pattern.compile ("(enum|set)\\((.*)\\)",
                                  Pattern.CASE_INSENSITIVE);
    Matcher m = pc.matcher (typeDef);
    // matches() fails unless it matches entire string
    if (m.matches ())
    {
      // split value list on commas, trim quotes from end of each word
      String[] v = m.group (2).split (",");
      for (int i = 0; i < v.length; i++)
        values.add (v[i].substring (1, v[i].length() - 1));
    }
    ctx.setAttribute (valListAttr, values);
  }

  %>
```

The function takes three arguments:

`ctx`
: The page context object.

`typeDefAttr`
: The name of the page attribute that contains the column definition. This is the function "input."

`valListAttr`
: The name of the page attribute into which to store the resulting array of legal column values. This is the function "output."

To generate a list element from the `size` column, begin by fetching the column metadata. Extract the column value list into a JSTL variable named `values` and the default value into a variable named `default` as follows:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT COLUMN_TYPE, COLUMN_DEFAULT
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE TABLE_SCHEMA = 'cookbook' AND TABLE_NAME = 'cow_order'
  AND COLUMN_NAME = 'size'
</sql:query>
<c:set var="typeDef" scope="page" value="${rs.rowsByIndex[0][0]}"/>
<% getEnumOrSetValues (pageContext, "typeDef", "values"); %>
<c:set var="defaultVal" scope="page" value="${rs.rowsByIndex[0][1]}"/>
```

Then use the value list and default value to construct a form element. For example, produce a set of radio buttons like this:

```
<c:forEach items="${values}" var="val">
  <input type="radio" name="size"
    value="<c:out value="${val}"/>"
```

```
    <c:if test="${val == defaultVal}">checked="checked"</c:if>
  /><c:out value="${val}"/><br />
</c:forEach>
```

or a pop-up menu like this:

```
<select name="size">
<c:forEach items="${values}" var="val">
  <option
    value="<c:out value="${val}"/>"
    <c:if test="${val == defaultVal}">selected="selected"</c:if>
  ><c:out value="${val}"/></option>
</c:forEach>
</select>
```

Don't Forget to HTML-Encode All List Content in Forms

The Ruby, PHP, and Python utility routines described in this recipe for generating list elements perform HTML-encoding of attribute values for the HTML tags that make up the list, such as the `name` and `value` attributes. They also encode the labels. I've noticed that many published accounts of list generation do not do this, or they encode the labels but not the values. That is a mistake. If either the label or the value contains a special character like `&` or `<`, the browser may misinterpret them, and your application will misbehave. It's also important to make sure that your encoding function turns double quotes into `&quot;` entities (or `&#34;`, which is equivalent), because tag attributes are so often enclosed within double quotes. Failing to convert a double quote to the entity name in an attribute value results in a double quote within a double-quoted string, which is malformed.

If you use the Perl CGI.pm module or the JSTL tags to produce HTML for form elements, encoding is taken care of for you. CGI.pm's form-related functions automatically perform encoding. Similarly, using the JSTL `<c:out>` tag to write attribute values from JSP pages produces properly encoded values.

The list-generating methods discussed here are not tied to any particular database table, so they can be used to create form elements for all kinds of data, not just those shown for the cow-ordering scenario. For example, to enable a user to pick a table name in a database administration application, generate a scrolling list that contains an item for each table in the database. A CGI.pm-based script might do so like this:

```
my $table_ref = $dbh->selectcol_arrayref (qq{
  SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'cookbook' ORDER BY TABLE_NAME
});
print scrolling_list (-name   => "table",
                      -values => $table_ref,
                      -size   => 10);     # display 10 items at a time
```

20.3. Creating Multiple-Pick Form Elements from Database Content

Problem

A form must present a field that offers several options and enables the user to select any number of them.

Solution

Use a multiple-pick list element, such as a set of checkboxes or a scrolling list.

Discussion

Multiple-pick form elements enable you to present multiple choices, any number of which can be selected, or possibly even none of them. For our example scenario in which customers order cow figurines online, the multiple-pick element is represented by the set of accessory items that are available. The `accessory` column in the `cow_order` table is represented as a `SET`, so the following statement returns the permitted and default values:

```
mysql> SELECT COLUMN_TYPE, COLUMN_DEFAULT FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA='cookbook' AND TABLE_NAME='cow_order'
    -> AND COLUMN_NAME='accessories';
+-------------------------------------------------+----------------+
| COLUMN_TYPE                                     | COLUMN_DEFAULT |
+-------------------------------------------------+----------------+
| set('cow bell','horns','nose ring','tail ribbon') | cow bell,horns |
+-------------------------------------------------+----------------+
```

The values listed in the definition can be reasonably represented as a set of checkboxes or a multiple-pick scrolling list. Either way, the `cow bell` and `horns` items should be selected initially because each is present in the column's default value. The following discussion shows the HTML syntax for these elements, then describes how to generate them from within scripts.

The material in this section relies heavily on Recipe 20.2, which discusses radio buttons, pop-up menus, and single-pick scrolling lists. I assume that you've already read that section.

Checkboxes

A group of checkboxes is similar to a group of radio buttons in that it consists of `<input>` elements that all have the same `name` attribute. However, the `type` attribute is `checkbox` rather than `radio`, and you can specify `checked` for as many items in

the group as you want selected by default. If no items are marked as `checked`, none are selected initially. The following checkbox set shows the cow accessory items with the first two items selected by default:

```
<input type="checkbox" name="accessories" value="cow bell"
  checked="checked" />cow bell
<input type="checkbox" name="accessories" value="horns"
  checked="checked" />horns
<input type="checkbox" name="accessories" value="nose ring" />nose ring
<input type="checkbox" name="accessories" value="tail ribbon" />tail ribbon
```

Scrolling lists

A multiple-pick scrolling list has most syntax in common with its single-pick counterpart. The differences are that you include a `multiple` attribute in the opening `<select>` tag, and default value selection differs. For a single-pick list, add `selected` to at most one item; in the absence of an explicit `selected` attribute, the first item is selected by default. For a multiple-pick list, add a `selected` attribute to as many of the items as you like; in the absence of `selected` attributes, no items are selected by default.

Represented as a multiple-pick scrolling list with `cow bell` and `horns` selected initially, the set of cow accessories looks like this:

```
<select name="accessories" size="3" multiple="multiple">
<option value="cow bell" selected="selected">cow bell</option>
<option value="horns" selected="selected">horns</option>
<option value="nose ring">nose ring</option>
<option value="tail ribbon">tail ribbon</option>
</select>
```

In CGI.pm-based Perl scripts, create checkbox sets or scrolling lists by invoking `checkbox_group()` or `scrolling_list()`. These functions take `name`, `values`, `labels`, and `default` arguments, just like their single-pick cousins. But because multiple items can be selected initially, CGI.pm permits the `default` argument to be specified as either a scalar value or a reference to an array of values. It also accepts the argument name `defaults` as a synonym for `default`.

To get the list of legal values for a `SET` column, do the same thing as in Recipe 20.2 for `ENUM` columns—invoke a utility routine that returns the column metadata:

```
my $acc_info = get_enumorset_info ($dbh, "cookbook", "cow_order", "accessories");
```

However, the default value for a `SET` column is not in a form that is directly usable for form element generation. MySQL represents `SET` default values as a list of zero or more items, separated by commas; for example, the default for the `accessories` column is `cow bell,horns`. That doesn't match the list-of-values format that the CGI.pm functions expect, so it's necessary to split the default value at the commas to obtain an array. The following expression shows how, taking into account the possibility that the default column value might be `undef` (`NULL`):

```
my @acc_def = defined ($acc_info->{default})
              ? split (/,/, $acc_info->{default})
              : ();
```

After splitting the default value, pass the resulting array by reference to the list-generating function you want to use:

```
print checkbox_group (-name      => "accessories",
                      -values    => $acc_info->{values},
                      -default   => \@acc_def,
                      -linebreak => 1);   # display buttons vertically

print scrolling_list (-name     => "accessories",
                      -values   => $acc_info->{values},
                      -default  => \@acc_def,
                      -size     => 3,     # display 3 items at a time
                      -multiple => 1);    # create multiple-pick list
```

When you use `SET` values like this to create list elements, the values are displayed in the order they are listed in the column definition. To produce a different display order, sort the values appropriately.

For Ruby, PHP, and Python, we can create utility functions to generate multiple-pick items. They have the following invocation syntax:

```
make_checkbox_group (name, values, labels, default, vertical)
make_scrolling_list (name, values, labels, default, size, multiple)
```

The `name`, `values`, and `labels` arguments to these functions are similar to those of the single-pick utility routines described in Recipe 20.2. `make_checkbox_group()` takes a `vertical` argument to indicate whether to stack the items vertically rather than horizontally. `make_scrolling_list()` was already described in Recipe 20.2 for producing single-pick lists. To use it here, the `multiple` argument should be true to produce a multiple-pick list. For both functions, the `default` argument can be an array of multiple values if several items should be selected initially.

`make_checkbox_group()` looks like this (shown here in Ruby; the PHP and Python versions are similar):

```
def make_checkbox_group(name, values, labels, default, vertical)
  # make sure default is an array (converts a scalar to an array)
  default = [ default ].flatten
  str = ""
  for i in 0...values.length do
    # select the item if it corresponds to one of the default values
    checked = (default.include?(values[i]) ? " checked=\"checked\"" : "")
    str << sprintf(
             "<input type=\"checkbox\" name=\"%s\" value=\"%s\"%s />%s",
             CGI.escapeHTML(name.to_s),
             CGI.escapeHTML(values[i].to_s),
             checked,
             CGI.escapeHTML(labels[i].to_s))
```

```
    str << "<br />" if vertical   # display items vertically
  end
  return str
end
```

To fetch the cow accessory information and present it using checkboxes, do this:

```
acc_info = get_enumorset_info(dbh, "cookbook", "cow_order", "accessories")
if acc_info["default"].nil?
  acc_def = []
else
  acc_def = acc_info["default"].split(",")
end

form << make_checkbox_group("accessories",
                            acc_info["values"],
                            acc_info["values"],
                            acc_def,
                            true)      # display items vertically
```

To display a scrolling list instead, invoke `make_scrolling_list()`:

```
form << make_scrolling_list("accessories",
                            acc_info["values"],
                            acc_info["values"],
                            acc_def,
                            3,         # display 3 items at a time
                            true)      # create multiple-pick list
```

In PHP, fetch the accessory information:

```
$acc_info = get_enumorset_info ($dbh, "cookbook", "cow_order", "accessories");
$acc_def = explode (",", $acc_info["default"]);
```

Then present checkboxes or a scrolling list:

```
print (make_checkbox_group ("accessories[]",
                            $acc_info["values"],
                            $acc_info["values"],
                            $acc_def,
                            TRUE));   # display items vertically

print (make_scrolling_list ("accessories[]",
                            $acc_info["values"],
                            $acc_info["values"],
                            $acc_def,
                            3,        # display 3 items at a time
                            TRUE));   # create multiple-pick list
```

Note that the field name in the PHP examples is specified as `accessories[]` rather than as `accessories`. In PHP, to permit a field to have multiple values, you must add `[]` to the name. If you omit the `[]`, the user can select multiple items while filling in the form, but PHP returns only one to your script. This issue comes up again (see Recipe 20.5) when we discuss how to process the contents of submitted forms.

In Python, to fetch the cow accessory information and present it using checkboxes or a scrolling list, do this:

```
acc_info = get_enumorset_info(conn, 'cookbook', 'cow_order', 'accessories')
if acc_info['default'] is None:
  acc_def = ""
else:
  acc_def = acc_info['default'].split(',')

print(make_checkbox_group('accessories',
                          acc_info['values'],
                          acc_info['values'],
                          acc_def,
                          True))  # display items vertically

print(make_scrolling_list('accessories',
                          acc_info['values'],
                          acc_info['values'],
                          acc_def,
                          3,      # display 3 items at a time
                          True))  # create multiple-pick list
```

In JSP pages, the `getEnumOrSetValues()` function used earlier to get the value list for the `size` column (an `ENUM`) can also be used for the `accessory` column (a `SET`). The column definition and default value can be obtained from `INFORMATION_SCHEMA`. Query the `COLUMNS` table, parse the type definition into a list of values named `values`, and put the default value in `defList` like this:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT COLUMN_TYPE, COLUMN_DEFAULT
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE TABLE_SCHEMA = 'cookbook'
  AND TABLE_NAME = 'cow_order'
  AND COLUMN_NAME = 'accessories'
</sql:query>
<c:set var="typeDef" scope="page" value="${rs.rowsByIndex[0][0]}"/>
<% getEnumOrSetValues (pageContext, "typeDef", "values"); %>
<c:set var="defList" scope="page" value="${rs.rowsByIndex[0][1]}"/>
```

For a `SET` column, the `defList` value might contain multiple values, separated by commas. It needs no special treatment; the JSTL `<c:forEach>` tag can iterate over such a string, so initialize the default values for a checkbox set as follows:

```
<c:forEach items="${values}" var="val">
  <input type="checkbox" name="accessories"
    value="<c:out value="${val}"/>"
    <c:forEach items="${defList}" var="defaultVal">
      <c:if test="${val == defaultVal}">checked="checked"</c:if>
    </c:forEach>
  /><c:out value="${val}"/><br />
</c:forEach>
```

For a multiple-pick scrolling list, do this:

```
<select name="accessories" size="3" multiple="multiple">
<c:forEach items="${values}" var="val">
  <option
    value="<c:out value="${val}"/>"
    <c:forEach items="${defList}" var="defaultVal">
      <c:if test="${val == defaultVal}">selected="selected"</c:if>
    </c:forEach>
  ><c:out value="${val}"/></option>
</c:forEach>
</select>
```

20.4. Loading Database Content into a Form

Problem

You want to display a form but initialize it using the contents of a database record, to present a record-editing form.

Solution

Generate the form as you usually would, but populate it with database content. That is, instead of setting the form fields to their usual defaults, set them to values retrieved from the database.

Discussion

The examples in earlier recipes that show how to generate form fields have either supplied no default value or have used the default value as specified in an `ENUM` or `SET` column definition. That's appropriate for presenting a "blank" form that you expect the user to fill in. However, for applications that present a web-based interface for record editing, it's more likely that you'd want to fill in the form using the content of an existing record for the initial values. This section discusses how to do that.

The examples shown here illustrate how to generate an editing form for rows from the `cow_order` table. Normally, you would permit the user to specify which record to edit. For simplicity, assume the use of a record that has an `id` value of `1`, with the following contents:

```
mysql> SELECT * FROM cow_order WHERE id = 1\G
*************************** 1. row ***************************
         id: 1
      color: Black & White
       size: large
accessories: cow bell,nose ring
  cust_name: Farmer Brown
cust_street: 123 Elm St.
```

```
 cust_city: Bald Knob
cust_state: AR
```

To generate a form with contents that correspond to a database record, use the column values for the element defaults as follows:

- For `<input>` elements such as radio buttons or checkboxes, add a `checked` attribute to each list item that matches the column value.
- For `<select>` elements such as pop-up menus or scrolling lists, add a `selected` attribute to each list item that matches the column value.
- For text fields represented as `<input>` elements of type `text`, set the `value` attribute to the corresponding column value. For example, to present a 60-character field for `cust_name`, initialized to `Farmer Brown`, do this:

  ```
  <input type="text" name="cust_name" value="Farmer Brown" size="60" />
  ```

  To present a `<textarea>` element instead, set the body to the column value. To create a field 40 columns wide and 3 rows high, write it like this:

  ```
  <textarea name="cust_name" cols="40" rows="3">
  Farmer Brown
  </textarea>
  ```

- In a record-editing situation, it's a good idea to include a unique value in the form so that you can tell which record the form contents represent when the user submits it. Use a hidden field to do this. Its value is not displayed to the user, but the browser returns it with the rest of the field values. Our sample record has an `id` column with a value of `1`, so the hidden field looks like this:

  ```
  <input type="hidden" name="id" value="1" />
  ```

The following examples show how to produce a form with `id` represented as a hidden field, `color` as a pop-up menu, `size` as a set of radio buttons, and `accessories` as a set of checkboxes. The customer information values are represented as text input boxes, except that `cust_state` is a single-pick scrolling list. You could make other choices, of course, such as to present the sizes as a pop-up menu rather than as radio buttons.

The `recipes` distribution scripts for the examples in this section are named *cow_edit.pl*, *cow_edit.jsp*, and so forth. Note that these scripts are designed only to present the entry form; they do nothing with the form contents when you click the Submit button.

The following procedure outlines how to load the sample `cow_table` record into an editing form for a CGI.pm-based Perl script:

1. Retrieve the column values for the record to load into the form:

   ```
   my $id = 1;     # select record number 1
   my ($color, $size, $accessories,
       $cust_name, $cust_street, $cust_city, $cust_state)
         = $dbh->selectrow_array (qq{
   ```

```
        SELECT
           color, size, accessories,
           cust_name, cust_street, cust_city, cust_state
         FROM cow_order WHERE id = ?
      }, undef, $id);
```

2. Begin the form:

```
print start_form (-action => url ());
```

3. Generate the hidden field containing the `id` value that uniquely identifies the `cow_order` record:

```
print hidden (-name => "id", -value => $id, -override => 1);
```

   The `override` argument forces CGI.pm to use the value specified in the `value` argument as the hidden field value. If `override` is not true, CGI.pm normally tries to use values present in the script execution environment to initialize form fields, even if you provide values in the field-generating calls. (CGI.pm does this to make it easier to redisplay a form with the values the user just submitted. For example, if you find that a form has been filled in incorrectly, you can redisplay it and ask the user to correct any problems. To make sure that a form element contains the value you specify, it's necessary to override this behavior.)

4. Create the fields that describe the cow figurine specifications. Generate these fields the same way as described in Recipes 20.2 and 20.3, except set the default values from the contents of record 1. The code here presents `color` as a pop-up menu, `size` as a set of radio buttons, and `accessories` as a set of checkboxes. Note that it splits the `accessories` value at commas to produce an array of values because the column value might name several accessory items:

```
my $color_ref = $dbh->selectcol_arrayref (qq{
  SELECT color FROM cow_color ORDER BY color
});

print br (), "Cow color:", br ();
print popup_menu (-name => "color",
                  -values => $color_ref,
                  -default => $color,
                  -override => 1);

my $size_info = get_enumorset_info ($dbh, "cookbook",
                                    "cow_order", "size");

print br (), "Cow figurine size:", br ();
print radio_group (-name => "size",
                   -values => $size_info->{values},
                   -default => $size,
                   -override => 1,
                   -linebreak => 1);

my $acc_info = get_enumorset_info ($dbh, "cookbook",
```

```
                                    "cow_order", "accessories");
my @acc_val = defined ($accessories)
              ? split (/,/, $accessories)
              : ();

print br (), "Cow accessory items:", br ();
print checkbox_group (-name => "accessories",
                      -values => $acc_info->{values},
                      -default => \@acc_val,
                      -override => 1,
                      -linebreak => 1);
```

5. Create the customer information fields. These are text input fields, except the state, shown here as a single-pick scrolling list:

```
print br (), "Customer name:", br ();
print textfield (-name => "cust_name",
                 -value => $cust_name,
                 -override => 1,
                 -size => 60);

print br (), "Customer street address:", br ();
print textfield (-name => "cust_street",
                 -value => $cust_street,
                 -override => 1,
                 -size => 60);

print br (), "Customer city:", br ();
print textfield (-name => "cust_city",
                 -value => $cust_city,
                 -override => 1,
                 -size => 60);

my @state_values;
my %state_labels;
my $sth = $dbh->prepare (qq{
  SELECT abbrev, name FROM states ORDER BY name
});
$sth->execute ();
while (my ($abbrev, $name) = $sth->fetchrow_array ())
{
  push (@state_values, $abbrev);  # save each value in an array
  $state_labels{$abbrev} = $name; # map each value to its label
}

print br (), "Customer state:", br ();
print scrolling_list (-name => "cust_state",
                      -values => \@state_values,
                      -labels => \%state_labels,
                      -default => $cust_state,
                      -override => 1,
                      -size => 6);      # display 6 items at a time
```

6. Create a form submission button and terminate the form:

```
print br (),
      submit (-name => "choice", -value => "Submit Form"),
      end_form ();
```

The same general procedure applies to other APIs. For example, in a JSP page, fetch the record to be edited and extract its contents into scalar variables like this:

```
<c:set var="id" value="1"/>
<sql:query dataSource="${conn}" var="rs">
  SELECT
    id, color, size, accessories,
    cust_name, cust_street, cust_city, cust_state
  FROM cow_order WHERE id = ?
  <sql:param value="${id}"/>
</sql:query>

<c:set var="row" value="${rs.rows[0]}"/>
<c:set var="id" value="${row.id}"/>
<c:set var="color" value="${row.color}"/>
<c:set var="size" value="${row.size}"/>
<c:set var="accessories" value="${row.accessories}"/>
<c:set var="cust_name" value="${row.cust_name}"/>
<c:set var="cust_street" value="${row.cust_street}"/>
<c:set var="cust_city" value="${row.cust_city}"/>
<c:set var="cust_state" value="${row.cust_state}"/>
```

Then use the values to initialize the various form elements, such as:

- The hidden field for the ID value:

```
<input type="hidden" name="id" value="<c:out value="${id}"/>"/>
```

- The `color` pop-up menu:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT color FROM cow_color ORDER BY color
</sql:query>
<br />Cow color:<br />
<select name="color">
<c:forEach items="${rs.rows}" var="row">
  <option
    value="<c:out value="${row.color}"/>"
    <c:if test="${row.color == color}">selected="selected"</c:if>
  ><c:out value="${row.color}"/></option>
</c:forEach>
</select>
```

- The `cust_name` text field:

```
<br />Customer name:<br />
<input type="text" name="cust_name"
```

```
value="<c:out value="${cust_name}"/>"
size="60" />
```

For Ruby, PHP, or Python, create the form using the utility functions developed in Recipes 20.2 and 20.3. See the *cow_edit.rb*, *cow_edit.php*, and *cow_edit.py* scripts for details.

20.5. Collecting Web Input

Problem

You want to extract input parameters submitted as part of a form or specified at the end of a URL.

Solution

Use the capabilities of your API that provide a means of accessing names and values of the input parameters in the execution environment of a web script.

Discussion

Earlier recipes in this chapter discuss how to retrieve information from MySQL and use it to generate various forms of output, such as static text, lists, hyperlinks, or form elements. In this recipe, we discuss the opposite problem—how to collect input from the Web. Applications for such input are many. For example, you can use the techniques shown here to extract the contents of a form submitted by a user. You might interpret the information as a set of search keywords, and then run a query against a product catalog to show the matching items to a customer. In this case, you use the Web to collect information from which to determine the client's interests. From that, construct an appropriate search query and display the results. If a form represents a survey, a mailing list sign-up sheet, or a poll, you might just store the values, using the data to create a new database record (or perhaps to update an existing record).

A script that receives input over the Web and uses it to interact with MySQL generally processes the information in a series of stages:

1. Extract the input from the execution environment. When a request arrives that contains input parameters, the web server places the input into the environment of the script that handles the request, and the script queries its environment to obtain the parameters. It may be necessary to decode special characters in the parameters to recover the actual values submitted by the client, if the extraction mechanism provided by your API doesn't do it for you. (For example, you might need to convert `%20` to space.)

2. Validate the input to make sure that it's legal. You cannot trust users to send legal values, so check input parameters to make sure they look reasonable. For example, if you expect a user to enter a number into a field, check the value to be sure that it's really numeric. If a form contains a pop-up menu constructed using the permitted values of an `ENUM` column, you might expect the value actually returned to be one of them. But there's no way to be sure except to check. Remember, you don't even know there is a real user on the other end of the network connection. It might be a malicious script roving the Web, trying to hack your site by exploiting weaknesses in your form-processing code.

   If you don't check input, you run the risk of storing garbage in your database or corrupting existing content. It is true that you can prevent entry of values that are invalid for the data types in your table columns by enabling strict SQL mode (see Recipe 12.1). However, there might be additional semantic constraints on what your application considers legal, in which case it's still useful to check values in your script before attempting to enter them. Also, by performing checks in your script, you may be able to present more meaningful error messages to users about problems in the input than the messages returned by the MySQL server when it detects bad data. For these reasons, it might be best to consider strict SQL mode a valuable additional level of protection, not necessarily sufficient in itself. That is, combine strict mode on the server side with client-side validation.
3. Construct an SQL statement based on the input. Typically, input parameters are used to add a record to a database, or to retrieve information from the database for display to the client. Either way, you use the input to construct a statement and send it to the MySQL server. Statement construction based on user input should be done with care, using proper escaping to avoid creating malformed or dangerous SQL statements. Use of placeholders is a good idea here (see Recipe 2.5).

The rest of this recipe explores the first of these three stages of input processing (pulling input from the execution environment). Recipes 20.6 and 20.7 cover the second and third stages. The first stage has little to do with MySQL, but is covered here because it's how you obtain the information used in the later stages.

Input obtained over the Web can be received in several ways, two of which are most common:

- As part of a `get` request, in which case input parameters are appended to the end of the URL. For example, the following URL invokes a PHP script named *price_quote.php* and specifies `item` and `quantity` parameters with values `D-0214` and `60`:

  ```
  http://localhost/mcb/price_quote.php?item=D-0214&quantity=60
  ```

  Such requests are generated when a user selects a hyperlink or submits a form that specifies `method="get"` in the `<form>` tag. A parameter list in a URL begins

with `?` and consists of *`name=value`* pairs separated by `;` or `&` characters. (It's also possible to place information in the middle of a URL, but this book doesn't cover that.)

- As part of a `post` request, such as a form submission that specifies `method="post"` in the `<form>` tag. The contents of a form for a `post` request are sent as parameters in the body of the request, rather than at the end of the URL.

You may also have occasion to process other types of input, such as uploaded files. Those are sent using `post` requests, but as part of a special kind of form element. Recipe 20.8 discusses file uploads.

When you gather input for a web script, you should consider how the input was sent. (Some APIs distinguish between input sent via `get` and `post`; others do not.) However, after you have pulled out the information that was sent, the request method doesn't matter. The validation and statement construction stages need not know whether parameters were sent using `get` or `post`.

The `recipes` distribution includes scripts that process input parameters in the *apache/params* directory (*tomcat/mcb* for JSP). Each script enables you to submit `get` or `post` requests, and shows how to extract and display the parameter values thus submitted. Examine these scripts to see how the parameter-extraction methods for the various APIs are used. Utility routines invoked by the scripts can be found in the library modules in the *lib* directory of the distribution.

Web input extraction conventions

To obtain input parameters passed to a script, familiarize yourself with your API's conventions so that you know what it does for you, and what you must do yourself. You should know the answers to these questions:

- How do you determine which parameters are available?
- How do you obtain a parameter value from the environment?
- Are values thus obtained the actual values submitted by the client, or do you need to decode them further?
- How are multiple-valued parameters handled (for example, when several items in a checkbox group are selected)?
- For parameters submitted in a URL, which separator character does the API expect between parameters? This may be `&` for some APIs and `;` for others. `;` is preferable as a parameter separator because it's not special in HTML like `&` is, but many browsers or other user agents separate parameters using `&`. If you construct a URL within a script that includes parameters at the end, be sure to use a parameter-separator character that the receiving script understands.

Perl. The Perl CGI.pm module makes input parameters available to scripts through the `param()` function. `param()` provides access to input submitted via either `get` or `post`, which simplifies your task as the script writer. You need not know which method a form used for submitting parameters. You need not perform any decoding, either; `param()` handles that as well.

To obtain a list of names of all available parameters, call `param()` with no arguments:

```
@param_names = param ();
```

To obtain the value of a specific parameter, pass its name to `param()`. In scalar context, `param()` returns the parameter value if it is single-valued, the first value if it is multiple-valued, or `undef` if the parameter is not available. In array context, `param()` returns all the parameter's values as a list, which is empty if the parameter is not available:

```
$id = param ("id");            # return scalar value
@options = param ("options");  # return list
```

A parameter with a given name might not be available if the form field with that name was left blank, or if there isn't any field with that name. Note too that a parameter value may be defined but empty. For good measure, you may want to check both possibilities. For example, to check for an `age` parameter and assign a default value of `unknown` if the parameter is missing or empty, you can do this:

```
$age = param ("age");
$age = "unknown" if !defined ($age) || $age eq "";
```

CGI.pm understands both `;` and `&` as URL parameter separator characters.

Ruby. Ruby scripts that use the `cgi` module access web script parameters through the `cgi` object you create for generating HTML elements. Its `param` method returns a hash of parameter names and values; access this hash or get the parameter names as follows:

```
params = cgi.params
param_names = cgi.params.keys
```

The value of a particular parameter is available from the hash of parameter names and values or from the `cgi` object:

```
id = cgi.params["id"]
id = cgi["id"]
```

The two access methods differ slightly. The `params` method returns each parameter value as an array. The array contains multiple entries if the parameter has multiple values, and is empty if the parameter is not present. The `cgi` object returns a single string. If the parameter has multiple values, only the first is returned. If the parameter is not present, the value is the empty string. For either access method, use the `has_key?` method to test whether a parameter is present.

The following listing shows how to get the parameter names and loop through each parameter to print its name and value, printing multiple-valued parameters as a comma-separated list:

```
params = cgi.params
param_names = cgi.params.keys
param_names.sort!
page << cgi.p { "Parameter names:" + param_names.join(", ") }

list = ""
param_names.each do |name|
  val = params[name]
  list << cgi.li {
    "type=#{val.class}, name=#{name}, value=" +
    CGI.escapeHTML(val.join(", "))
  }
end
page << cgi.ul { list }
```

The `cgi` module understands both `;` and `&` as URL parameter separator characters.

PHP. Input parameters are available to PHP several ways, depending on your configuration settings:

- If the `track_vars` variable is enabled (which it is by default), parameters are available in the `$_GET` and `$_POST` arrays. If a form contains a field named `id`, the value is available as `$_GET["id"]` or `$_POST["id"]`, depending on whether the form was submitted via `get` or `post`. `$_GET` and `$_POST` are "superglobal" arrays (accessible in any scope without being declared as global).
- If the `register_globals` variable is enabled, parameters are assigned to global variables of the same name. In this case, the value of a field named `id` is available as the variable `$id`, regardless of whether the request was sent via `get` or `post`. It's dangerous to rely on this variable, for reasons described shortly. PHP scripts in this book do not rely on `register_globals` (which in any case is deprecated in PHP 5.3 and removed in 5.4). Instead, they obtain input through the global parameter arrays.

The `track_vars` and (if present) `register_globals` settings can be compiled into PHP or configured in the PHP *php.ini* file. As mentioned previously, `track_vars` is enabled by default, so I'll assume that this is true for your PHP installation.

`register_globals` was designed to make it convenient to access input parameters through global variables, but it poses a security risk and is therefore best disabled in versions of PHP that have it. Suppose that you write a script that requires the user to supply a password, represented by the `$password` variable. In the script, you might check the password like this:

```
if (check_password ($password))
  $password_is_ok = 1;
```

The intent here is that if the password matches, the script sets `$password_is_ok` to `1`. Otherwise, it leaves `$password_is_ok` unset (which compares false in Boolean expressions). But suppose that someone invokes your script as follows:

```
http://your.host.com/chkpass.php?password_is_ok=1
```

If `register_globals` is enabled, PHP sees that the `password_is_ok` parameter is set to `1` and sets the corresponding `$password_is_ok` variable to `1`. The result is that when your script executes, `$password_is_ok` is `1` no matter what password was given, or even if *no* password was given! Thus, `register_globals` enables outside users to supply default values for global variables in your scripts. The best solution is to disable `register_globals` and check the global arrays (`$_GET`, `$_POST`) for input parameter values. If you cannot disable `register_globals`, take care not to assume that PHP variables have no value initially. Unless you expect a global variable to be set from an input parameter, initialize it explicitly to a known value. The password-checking code should be written as follows to make sure that only `$password` (and not `$password_is_ok`) can be set from an input parameter. That way, `$password_is_ok` is assigned a value by the script itself whatever the result of the test:

```
$password_is_ok = 0;
if (check_password ($password))
  $password_is_ok = 1;
```

Another complicating factor when retrieving input parameters in PHP is that they may need some decoding, depending on the value of the `magic_quotes_gpc` configuration variable (if present; like `register_globals`, `magic_quotes_gpc` is deprecated in PHP 5.3 and removed in 5.4). If magic quotes are enabled, any quote, backslash, and NUL characters in input parameter values accessed by your scripts will be escaped with backslashes. I suppose that the intent is to save you a step by permitting you to extract values and use them directly in SQL statement strings. However, that's only useful if you plan to use web input in a statement with no preprocessing or validity checking, which is dangerous. You should check your input first, in which case it's necessary to strip out the slashes, anyway. This means that having magic quotes turned on isn't really very useful.

Given the various sources through which input parameters may be available, and the fact that they may or may not contain extra backslashes, extracting input in PHP scripts can be an interesting problem. If you have control of your server and can set the values of the various configuration settings, you can of course write your scripts based on those settings. But if you do not control your server or are writing scripts that need to run on several machines, you may not know in advance what the settings are. Fortunately, it's possible to write reasonably general-purpose parameter-extraction code that works correctly with few assumptions about your PHP operating environment. The following utility function, `get_param_val()`, takes a parameter name as its argument and returns

the corresponding parameter value. If the parameter is not available, the function returns an unset value. (`get_param_val()` uses a helper function, `strip_slash_helper()`, which is discussed shortly.)

```
function get_param_val ($name)
{
  $val = NULL;
  if (isset ($_GET[$name]))
    $val = $_GET[$name];
  else if (isset ($_POST[$name]))
    $val = $_POST[$name];
  if (isset ($val) && get_magic_quotes_gpc ())
    $val = strip_slash_helper ($val);
  return ($val);
}
```

To use this function to obtain the value of a single-valued parameter named `id`, call it like this:

```
$id = get_param_val ("id");
```

Test `$id` to determine whether the `id` parameter was present in the input:

```
if (isset ($id))
  ... id parameter is present ...
else
  ... id parameter is not present ...
```

For a form field that might have multiple values (such as a checkbox group or a multiple-pick scrolling list), represent it in the form using a name that ends in `[]`. For example, a list element constructed from the `SET` column `accessories` in the `cow_order` table has one item for each permitted set value. To make sure PHP treats the element value as an array, name the field `accessories[]`, not `accessories`. (See Recipe 20.3 for an example.) When the form is submitted, PHP places the array of values in a parameter named without the `[]`. To access it, do this:

```
$accessories = get_param_val ("accessories");
```

The value of the `$accessories` variable is an array, whether the parameter has multiple values, a single value, or even no values. The determining factor is not whether the parameter actually *has* multiple values, but whether you named the corresponding field in the form using `[]` notation.

The `get_param_val()` function checks the `$_GET` and `$_POST` arrays for parameter values. Thus, it works correctly regardless of whether the request was made by `get` or `post`, or whether `register_globals` is enabled. It assumes only that `track_vars` is enabled.

`get_param_val()` also works correctly regardless of whether magic quoting is enabled. It uses a helper function `strip_slash_helper()` that performs backslash stripping from parameter values if necessary:

```
function strip_slash_helper ($val)
{
  if (!is_array ($val))
    $val = stripslashes ($val);
  else
  {
    foreach ($val as $k => $v)
      $val[$k] = strip_slash_helper ($v);
  }
  return ($val);
}
```

`strip_slash_helper()` checks whether a value is a scalar or an array and processes it accordingly. It uses a recursive algorithm for array values because in PHP it's possible to create nested arrays from input parameters.

To make it easy to obtain a list of all parameter names, use another utility function:

```
function get_param_names ()
{
  # construct an array in which each element has a parameter name as
  # both key and value.  (Using names as keys eliminates duplicates.)
  $names = array ();
  foreach (array_keys ($_GET) as $name)
    $names[$name] = $name;
  foreach (array_keys ($_POST) as $name)
    $names[$name] = $name;
  return ($names);
}
```

`get_param_names()` returns a list of parameter names present in the HTTP variable arrays, with duplicate names removed if there is overlap between the arrays. The return value is an array with its keys and values both set to the parameter names. This way you can use either the keys or the values as the list of names. The following example prints the names, using the values:

```
$param_names = get_param_names ();
foreach ($param_names as $name)
  print (htmlspecialchars ($name) . "<br />");
```

To construct URLs that point to PHP scripts and that have parameters at the end, separate the parameters by `&` characters. To use a different character (such as `;`), change the separator by setting the `arg_separator` configuration variable in the PHP initialization file.

Python. The Python `cgi` module provides access to input parameters that are present in the script environment. Import that module, then create a `FieldStorage` object:

```
import cgi

params = cgi.FieldStorage()
```

The `FieldStorage` object contains information for parameters submitted via either `get` or `post` requests, so you need not know which method was used to send the request. The object also contains an element for each parameter present in the environment. Its `key()` method returns a list of available parameter names:

```
param_names = params.keys()
```

If a given parameter, `name`, is single-valued, the value associated with it is a scalar that you can access as follows:

```
val = params[name].value
```

If the parameter is multiple-valued, `params[name]` is a list of `MiniFieldStorage` objects that have `name` and `value` attributes. Each has the same name (it will be equal to `name`) and one of the parameter's values. To create a list containing all the values for such a parameter, do this:

```
val = []
for item in params[name]:
  val.append(item.value)
```

To avoid having to distinguish whether a parameter has a single value or multiple values, use `getlist()`. The following listing shows how to get the parameter names and loop through each parameter to print its name and value, printing multiple-valued parameters as a comma-separated list:

```
params = cgi.FieldStorage()
param_names = params.keys()
param_names.sort()
print("<p>Parameter names: %s</p>" % param_names)
items = []
for name in param_names:
  val = ','.join(params.getlist(name))
  items.append("name=" + name + ", value=" + val)
print(make_unordered_list(items))
```

Python raises an exception if you try to access a parameter not present in the `FieldStorage` object. To avoid this, use `has_key()` to find out whether the parameter exists:

```
if params.has_key(name):
  print("parameter %s exists" % name)
else:
  print("parameter %s does not exist" % name)
```

Single-valued parameters have attributes other than `value`. For example, a parameter representing an uploaded file has additional attributes you can use to get the file's contents. Recipe 20.8 discusses this further.

The `cgi` module expects URL parameters to be separated by `&` characters. To construct a hyperlink that points to a Python script based on the `cgi` module, don't separate the parameters by `;` characters.

Java. Within JSP pages, the implicit `request` object has the following methods for accessing the request parameters:

`getParameterNames()`
: Returns an enumeration of `String` objects, one for each parameter name present in the request.

`getParameterValues(String name)`
: Returns an array of `String` objects, one for each value associated with the parameter, or `null` if the parameter does not exist.

`getParameterValue(String name)`
: Returns the first value associated with the parameter, or `null` if the parameter does not exist.

The following example shows one way to use these methods to display request parameters:

```
<%@ page import="java.util.*" %>

<ul>
<%
  Enumeration e = request.getParameterNames ();
  while (e.hasMoreElements ())
  {
    String name = (String) e.nextElement ();
    // use array in case parameter is multiple-valued
    String[] val = request.getParameterValues (name);
    out.println ("<li> name: " + name + "; values:");
    for (int i = 0; i < val.length; i++)
      out.println (val[i]);
    out.println ("</li>");
  }
%>
</ul>
```

Request parameters are also available within JSTL tags, using the special variables `param` and `paramValues`. `param[name]` returns the first value for a given parameter and thus is most suited for single-valued parameters:

```
color value:
<c:out value="${param['color']}"/>
```

`paramValues[name]` returns an array of values for the parameter, so it's useful for parameters that can have multiple values:

```
accessory values:
<c:forEach items="${paramValues['accessories']}" var="val">
  <c:out value="${val}"/>
</c:forEach>
```

If a parameter name is legal as an object property name, you can also access the parameter using dot notation:

```
color value:
<c:out value="${param.color}"/>
accessory values:
<c:forEach items="${paramValues.accessories}" var="val">
  <c:out value="${val}"/>
</c:forEach>
```

To produce a list of parameter objects with `key` and `value` attributes, iterate over the `paramValues` variable:

```
<ul>
  <c:forEach items="${paramValues}" var="p">
    <li>
      name:
      <c:out value="${p.key}"/>;
      values:
      <c:forEach items="${p.value}" var="val">
        <c:out value="${val}"/>
      </c:forEach>
    </li>
  </c:forEach>
</ul>
```

To construct URLs that point to JSP pages and that have parameters at the end, separate the parameters by `&` characters.

20.6. Validating Web Input

Problem

After extracting the parameters supplied to a script, you want to check them to be sure they're valid.

Solution

Web input processing is one form of data import, so after you've extracted the input parameters, validate them using the techniques discussed in Chapter 12.

Discussion

One phase of web form processing is extracting the input returned when the user submits the form. It's also possible to receive input in the form of parameters at the end of a URL. Either way, if you plan to store the input in your database, it's important to check it to be sure that it's valid.

When clients send input to you over the Web, you don't really know what they're sending. If you present a form for users to fill out, most of the time they'll probably be nice and enter the kinds of values you expect. But a malicious user can save the form to a file, modify the file to permit form options you don't intend, reload the file into a browser window, and submit the modified form. Your form-processing script won't know the difference. If you write it only to process the kinds of values that well-intentioned users submit, the script may misbehave or crash when presented with unexpected input—or perhaps even do bad things to your database. (Recipe 20.7 discusses what kinds of bad things.) For this reason, it's prudent to perform some validity checking on web input before using it to construct SQL statements.

Preliminary checking is a good idea even for nonmalicious users. If a user neglects to provide a required value, you must present a reminder to supply one. The check might be simple ("Is the parameter present?") or more involved. Typical validation operations include the following:

- Checking content format, such as making sure a value looks like an integer or a date. This may involve some reformatting for acceptability to MySQL (for example, changing a date from *MM/DD/YY* to ISO format).
- Determining whether a value is a member of a legal set of values. Perhaps the value must be listed in the definition for an `ENUM` or `SET` column, or must be present in a lookup table.
- Filtering out extraneous characters such as spaces or dashes from telephone numbers or credit card numbers.

Some of these operations have little to do with MySQL, except in the sense that you want values to be appropriate for the types of the columns in which you store them or against which you match them. For example, before storing a value in an `INT` column, you can make sure that it's an integer first, using a test like this (shown here using Perl):

```
$val =~ /^\d+$/
  or die "Hey! '" . escapeHTML ($val) . "' is not an integer!\n";
```

For other types of validation, MySQL is intimately involved. If a field value is to be stored into an `ENUM` column, you can make sure the value is one of the legal enumeration values by checking the column definition in `INFORMATION_SCHEMA`.

Having described some of the kinds of web input validation you might want to carry out, I won't further discuss them here. Chapter 12 describes these and other forms of validation testing. That chapter is oriented largely toward bulk input validation, but the techniques discussed there apply to web programming as well.

20.7. Storing Web Input in a Database

Problem

Input obtained over the Web cannot be trusted and should not be entered into a database without taking the proper precautions.

Solution

Sanitize data values by using placeholders or a quoting function so that SQL statements you construct are valid and not subject to SQL injection attacks. Enable strict SQL mode so the MySQL server rejects values that are invalid for column data types.

Discussion

After you've extracted input parameter values in a web script and checked them to make sure they're valid, you're ready to use them to construct an SQL statement. This is actually the easy part of input processing, although it's necessary to take the proper precautions to avoid making a mistake that you'll regret. Let's consider what can go wrong, and then see how to prevent problems.

Suppose that a form acts as a frontend to a simple search engine and contains a keyword field. When a user submits a keyword, you intend to use it to find matching rows in a table by constructing a statement like this:

```
SELECT * FROM mytbl WHERE keyword = 'keyword_val'
```

Here, *keyword_val* represents the value entered by the user. If the value is something like `eggplant`, the resulting statement is:

```
SELECT * FROM mytbl WHERE keyword = 'eggplant'
```

The statement returns all eggplant-matching rows, presumably generating a small result set. But suppose that the user is tricky and tries to subvert your script by entering the following value:

```
eggplant' OR 'x'='x
```

In this case, the statement becomes:

```
SELECT * FROM mytbl WHERE keyword = 'eggplant' OR 'x'='x'
```

That statement matches every row in the table! If the table is quite large, the input effectively becomes a denial-of-service attack because it causes your system to divert resources away from legitimate requests into doing useless work. This type of attack is known as SQL injection because the user is injecting executable SQL code into your statement where you expect to receive only a nonexecutable data value. Likely results of SQL injection attacks include the following:

- Extra load on the MySQL server
- Out-of-memory problems in your script as it tries to digest the result set received from MySQL
- Extra network bandwidth consumption as the script sends the results to the client

If your script generates a `DELETE` or `UPDATE` statement, the consequences of this kind of subversion can be much worse. Your script might issue a statement that empties a table completely or changes all of its rows, when you intended to permit it to affect at most a single row.

Try to Break Your Web Scripts

The discussion in this section is phrased in terms of guarding against other users from attacking your scripts. But it's not a bad idea to put yourself in the place of an attacker and adopt the mindset, "How can I break this application?" That is, consider whether you can submit input that it won't handle and that causes it to generate a malformed statement. If you can cause the application to misbehave, so can other people, either deliberately or accidentally. Be wary of bad input, and write your applications accordingly. It's better to be prepared than just hope.

The implication of the preceding discussion is that providing a web interface to your database opens you up to certain forms of security vulnerabilities. However, you can prevent these problems by means of a simple precaution that you should already be following: don't put data values received from external sources literally into statement strings. Use placeholders or an encoding function instead. For example, handle an input parameter in Perl using a placeholder:

```
$sth = $dbh->prepare ("SELECT * FROM mytbl WHERE keyword = ?");
$sth->execute (param ("keyword"));
# ... fetch result set ...
```

Or by using `quote()`:

```
$keyword = $dbh->quote (param ("keyword"));
$sth = $dbh->prepare ("SELECT * FROM mytbl WHERE keyword = $keyword");
$sth->execute ();
# ... fetch result set ...
```

Either way, if the user enters the subversive value, the statement becomes harmless:

```
SELECT * FROM mytbl WHERE keyword = 'eggplant\' OR \'x\'=\'x'
```

As a result, the statement matches no rows rather than all rows—definitely a more suitable response to someone who's trying to break your script.

Recipe 2.5 discusses similar placeholder and quoting techniques for Ruby, PHP, Python, and Java. For JSP pages written using the JSTL tag library, quote input parameter values using placeholders and the `<sql:param>` tag. For example, to use the value of a form parameter named `keyword` in a `SELECT` statement, do this:

```
<sql:query dataSource="${conn}" var="rs">
  SELECT * FROM mytbl WHERE keyword = ?
  <sql:param value="${param['keyword']}"/>
</sql:query>
```

One issue not covered by placeholder techniques involves a question of interpretation: If a form field is optional, what should you store in the database if the user leaves the field empty? Perhaps the value represents an empty string—or perhaps it should be interpreted as `NULL`. One way to resolve this question is to consult the column metadata (see Recipe 10.6). If the column can contain `NULL` values, interpret an empty field as `NULL`. Otherwise, take an empty field to mean an empty string.

Placeholders and encoding functions apply only to SQL data values. They do not address how to handle web input used for other kinds of statement elements such as identifiers: names of databases, tables, and columns. If you intend to include such values into a statement literally, you should check them first. For example, to construct a statement such as the following, you should verify that `$tbl_name` contains a reasonable value:

```
SELECT * FROM $tbl_name;
```

But what does "reasonable" mean? If your tables don't have strange characters in their names, it may be sufficient to make sure that `$tbl_name` contains only alphanumeric characters or underscores. Alternatively, issue a statement that determines whether the table actually exists. (Check `INFORMATION_SCHEMA` or use `SHOW TABLES`.) This is more foolproof, at the cost of an additional statement.

A better option is to use an identifier-quoting routine, if you have one (see Recipe 2.6). This approach requires no extra statement because it renders any string safe for use in a statement. If the identifier does not exist, the statement simply fails as it should.

For additional protection in your web scripts, combine client-side checking of input values with strict server-side checking. You can set the server SQL mode to be restrictive about accepting input values so that it rejects values that don't match your table column data types. For discussion of the SQL mode and input value checking, see Recipe 12.1.

See Also

Several other recipes in this chapter illustrate how to incorporate web input into statements. Recipe 20.8 shows how to upload files and load them into MySQL. Recipe 20.9 demonstrates a simple search application using input as search keywords. Recipes 20.10 and 20.11 process parameters submitted via URLs.

20.8. Processing File Uploads

Problem

You want to permit files to be uploaded to your web server and stored in your database.

Solution

Present the user with a web form that includes a file field. When the user submits the form, extract the file and store it.

Discussion

One special kind of web input is an uploaded file. A file is sent as part of a `post` request, but it's handled differently from other `post` parameters because a file is represented by several pieces of information such as its contents, its MIME type, its original filename on the client, and its name in temporary storage on the web server host.

To handle file uploads, you must send a special kind of form to the user; this is true no matter what API you use to create the form. When the user submits the form, the operations that check for and process an uploaded file are API-specific.

To create a form that enables files to be uploaded, the opening `<form>` tag should specify the `post` method and must also include an `enctype` (encoding type) attribute with a value of `multipart/form-data`:

```
<form method="post" enctype="multipart/form-data" action="script_name">
```

If the form uses the `application/x-www-form-urlencoded` encoding type, file uploads will not work properly.

To include a file upload field in the form, use an `<input>` element of type `file`. For example, this element presents a 60-character file field named `upload_file`:

```
<input type="file" name="upload_file" size="60" />
```

The browser displays this field as a text input box into which the user can enter the name manually. It also displays a Browse button that enables the user to select the file via a standard file-browsing system dialog. When the user chooses a file and submits the form, the browser encodes the file contents for inclusion into the resulting `post` request. At that point, the web server receives the request and invokes your script to process it. The specifics vary for particular APIs, but file uploads generally work like this:

- The file will already have been uploaded and stored in a temporary directory by the time your upload-handling script begins executing. All your script has to do is read it. The temporary file is available to your script either as an open file descriptor or the temporary filename, or perhaps both. The file size can be obtained through the

file descriptor. The API may also make available other information about the file, such as its MIME type. (But note that some browsers may not send a MIME value.)

- The web server automatically deletes uploaded files when your script terminates. If you want a file's contents to persist beyond the end of your script's execution, the script must save the file to a more permanent location, such as in a database or somewhere else in the filesystem. If you save the file in the filesystem, the directory where you store it must be accessible to the web server. (Don't put it under the document root or any user home directories. That effectively enables a remote attacker to install scripts and HTML files on your web server.)
- The API might enable you to control the location of the temporary file directory or the maximum size of uploaded files. Changing the directory to one that is accessible only to your web server may improve security against local exploits by other users with login accounts on the server host.

This recipe discusses how to create forms that include a file upload field. It also demonstrates how to handle uploads using a Perl script, *post_image.pl*. The script is somewhat similar to the *store_image.pl* script for loading images from the command line (see Recipe 19.6). *post_image.pl* differs in that it enables you to store images over the Web by uploading them, and it stores images only in MySQL, whereas *store_image.pl* stores them in both MySQL and the filesystem.

This recipe also discusses how to obtain file upload information using PHP and Python. It does not repeat the entire image-posting scenario shown for Perl, but the `recipes` distribution contains implementations equivalent to *post_image.pl* for the other languages.

Uploads in Perl

The CGI.pm module enables you to specify multipart encoding for a form several ways. The following statements are equivalent:

```
print start_form (-action => url (), -enctype => "multipart/form-data");

print start_form (-action => url (), -enctype => MULTIPART ());

print start_multipart_form (-action => url ());
```

The first statement specifies the encoding type literally. The second uses the CGI.pm `MULTIPART()` function, which is easier than trying to remember the literal encoding value. The third statement is easiest of all because `start_multipart_form()` supplies the `enctype` parameter automatically. (Like `start_form()`, `start_multipart_form()` uses a default request method of `post`, so you need not include a `method` argument.)

Here's a simple form that includes a text field that enables the user to assign a name to an image, a file field so that the user can select the image file, and a Submit button:

```
print start_multipart_form (-action => url ()),
      "Image name:", br (),
      textfield (-name =>"image_name", -size => 60),
      br (), "Image file:", br (),
      filefield (-name =>"upload_file", -size => 60),
      br (), br (),
      submit (-name => "choice", -value => "Submit"),
      end_form ();
```

When the user submits an uploaded file, begin processing it by extracting the parameter value for the file field:

```
$file = param ("upload_file");
```

The value for a file upload parameter is special in CGI.pm because you can use it two ways. You can treat it as an open file handle to read the file's contents or pass it to `uploadInfo()` to obtain a reference to a hash that provides information about the file such as its MIME type. The following listing shows how *post_image.pl* presents the form and processes a submitted form. When first invoked, *post_image.pl* generates a form with an upload field. For the initial invocation, no file will have been uploaded, so the script does nothing else. If the user submitted an image file, the script gets the image name, reads the file contents, determines its MIME type, and stores a new row in the `image` table. For illustrative purposes, *post_image.pl* also displays all the information that the `uploadInfo()` function makes available about the uploaded file:

```
#!/usr/bin/perl
# post_image.pl: enable user to upload image files using post requests

use strict;
use warnings;
use CGI qw(:standard escapeHTML);
use Cookbook;

print header (), start_html (-title => "Post Image");

# Use multipart encoding because the form contains a file upload field

print start_multipart_form (-action => url ()),
      "Image name:", br (),
      textfield (-name =>"image_name", -size => 60),
      br (), "Image file:", br (),
      filefield (-name =>"upload_file", -size => 60),
      br (), br (),
      submit (-name => "choice", -value => "Submit"),
      end_form ();

# Get a handle to the image file and the name to assign to the image

my $image_file = param ("upload_file");
my $image_name = param ("image_name");
```

```
# Must have either no parameters (in which case that script was just
# invoked for the first time) or both parameters (in which case the form
# was filled in).  If only one was filled in, the user did not fill in the
# form completely.

my $param_count = 0;
++$param_count if defined ($image_file) && $image_file ne "";
++$param_count if defined ($image_name) && $image_name ne "";

if ($param_count == 0)      # initial invocation
{
  print p ("No file was uploaded.");
}
elsif ($param_count == 1)   # incomplete form
{
  print p ("Please fill in BOTH fields and resubmit the form.");
}
else                        # a file was uploaded
{
  my ($size, $data);

  # If an image file was uploaded, print some information about it,
  # then save it in the database.

  # Get reference to hash containing information about file
  # and display the information in "key=x, value=y" format
  my $info_ref = uploadInfo ($image_file);
  print p ("Information about uploaded file:");
  foreach my $key (sort (keys (%{$info_ref})))
  {
    print p ("key="
             . escapeHTML ($key)
             . ", value="
             . escapeHTML ($info_ref->{$key}));
  }
  $size = (stat ($image_file))[7]; # get file size from file handle
  print p ("File size: " . $size);

  binmode ($image_file);  # helpful for binary data
  if (sysread ($image_file, $data, $size) != $size)
  {
    print p ("File contents could not be read.");
  }
  else
  {
    print p ("File contents were read without error.");

    # Get MIME type, use generic default if not present

    my $mime_type = $info_ref->{'Content-Type'};
    $mime_type = "application/octet-stream" unless defined ($mime_type);
```

```
        # Save image in database table.  (Use REPLACE to kick out any
        # old image that has the same name.)

        my $dbh = Cookbook::connect ();
        $dbh->do ("REPLACE INTO image (name,type,data) VALUES(?,?,?)",
                  undef,
                  $image_name, $mime_type, $data);
        $dbh->disconnect ();
    }
}

print end_html ();
```

Uploads in PHP

To write an upload form in PHP, include a file field. If you like, also include a hidden field preceding the file field that has a name of `MAX_FILE_SIZE` and a value of the largest file size you're willing to accept:

```
<form method="post" enctype="multipart/form-data"
  action="<?php print ($_SERVER["PHP_SELF"]); ?>">
<input type="hidden" name="MAX_FILE_SIZE" value="4000000" />
Image name:<br />
<input type="text" name="image_name" size="60" />
<br />
Image file:<br />
<input type="file" name="upload_file" size="60" />
<br /><br />
<input type="submit" name="choice" value="Submit" />
</form>
```

Be aware that `MAX_FILE_SIZE` is advisory only; it can be subverted easily. To specify a value that cannot be exceeded, use the `upload_max_filesize` configuration variable in the *php.ini* PHP configuration file. There is also a `file_uploads` variable that controls whether file uploads are permitted at all.

The `upload_tmp_dir` PHP configuration variable controls where uploaded files are saved. This is */tmp* by default on many systems, but you may want to override it to reconfigure PHP to use a different directory that's owned by the web server user and thus more private.

When the user submits the form, PHP places file upload information from `post` requests in an array, `$_FILES`, that has one entry for each uploaded file. `$_FILES` is a superglobal array (accessible in any scope without being declared as global). Each entry within the array is itself an array with four elements. For example, if a form has a file field named `upload_file` and the user submits a file, information about it is available in the following variables:

```
$_FILES["upload_file"]["name"]
$_FILES["upload_file"]["tmp_name"]
```

```
$_FILES["upload_file"]["size"]
$_FILES["upload_file"]["type"]
```

These variables represent the original filename on the client host, the temporary filename on the server host, the file size in bytes, and the file MIME type. Be careful here because there may be an entry for an upload field even if the user submitted no file. In this case, the `tmp_name` value will be the empty string or the string `none`.

To simplify access to file upload information, use a utility routine that does the work. The following function, `get_upload_info()`, takes an argument corresponding to the name of a file upload field. Then it examines the `$_FILES` array and returns an associative array of information about the file, or a `NULL` value if the information is not available. For a successful call, the array element keys are `"tmp_name"`, `"name"`, `"size"`, and `"type"`:

```
function get_upload_info ($name)
{
  # Check the $_FILES array tmp_name member to make sure there is a
  # file. (The entry might be present even if no file was uploaded.)
  $val = NULL;
  if (isset ($_FILES[$name])
      && $_FILES[$name]["tmp_name"] != ""
      && $_FILES[$name]["tmp_name"] != "none")
    $val = $_FILES[$name];
  return ($val);
}
```

See the *post_image.php* script for details about how to use this function to get image information and store it in MySQL.

Uploads in Python

In Python, write a simple upload form like this:

```
print('''
<form method="post" enctype="multipart/form-data" action="%s">
Image name:<br />
<input type="text" name="image_name", size="60" />
<br />
Image file:<br />
<input type="file" name="upload_file", size="60" />
<br /><br />
<input type="submit" name="choice" value="Submit" />
</form>
''' % (os.environ['SCRIPT_NAME']))
```

When the user submits the form, obtain its contents using the `FieldStorage()` method of the `cgi` module (see Recipe 20.5). The resulting object contains an element for each input parameter. For a file upload field, get this information as follows:

```
form = cgi.FieldStorage()
if form.has_key('upload_file') and form['upload_file'].filename != '':
```

```
  image_file = form['upload_file']
else:
  image_file = None
```

According to most of the documentation that I have read, the `file` attribute of an object that corresponds to a file field should be true if a file has been uploaded. Unfortunately, the `file` attribute seems to be true even when the user submits the form but leaves the file field blank. It may even be the case that the `type` attribute is set when no file actually was uploaded (for example, to `application/octet-stream`). In my experience, a more reliable way to determine whether a file was uploaded is to test the `filename` attribute:

```
form = cgi.FieldStorage()
if form.has_key('upload_file') and form['upload_file'].filename:
  print("<p>A file was uploaded</p>")
else:
  print("<p>A file was not uploaded</p>")
```

Assuming that a file was uploaded, access the parameter's `value` attribute to read the file and obtain its contents:

```
data = form['upload_file'].value
```

See the *post_image.py* script for details about how to use this function to get image information and store it in MySQL.

20.9. Performing Web-Based Database Searches

Problem

You want to implement a web-based search interface.

Solution

Present a form containing fields that enable the user to supply search parameters such as keywords. Use the submitted keywords to construct a database query, then display the query results.

Discussion

A script that implements a web-based search interface provides a convenience for people who visit your website because they need not know any SQL to find information in your database. Instead, visitors supply keywords that describe what they're interested in and your script figures out the appropriate statements to run on their behalf. A common paradigm for this activity involves a form containing one or more fields for entering search parameters. The user fills in the form, submits it, and receives back a new page containing the records that match the parameters.

As the writer of such a script, you must handle these operations:

1. Generate the form and send it to the users.
2. Interpret the submitted form and construct an SQL statement based on its contents. This includes proper use of placeholders or quoting to prevent bad input from crashing or subverting your script.
3. Execute the statement and display its result. This can be simple if the result set is small, or more complex if it is large. In the latter case, you may want to present the matching records using a paged display—that is, a display consisting of multiple pages, each of which shows a subset of the entire statement result. Multiple-page displays have the benefit of not overwhelming the user with huge amounts of information all at once. Recipe 20.10 discusses how to implement them.

This recipe demonstrates a script that implements a minimal search interface: a form with one keyword field, from which a statement is constructed that returns at most one record. The script performs a two-way search of the `states` table. That is, if the user enters a state name, it looks up the corresponding abbreviation. Conversely, if the user enters an abbreviation, it looks up the name. The script, *search_state.pl*, looks like this:

```
#!/usr/bin/perl
# search_state.pl: Simple "search for state" application.

# Present a form with an input field and a submit button.  User enters
# a state abbreviation or a state name into the field and submits the
# form.  Script finds the abbreviation and displays the full name, or
# vice versa.

use strict;
use warnings;
use CGI qw(:standard escapeHTML);
use Cookbook;

my $title = "State Name or Abbreviation Lookup";

print header (), start_html (-title => $title);

# If keyword parameter is present and nonempty, perform a lookup.

my $keyword = param ("keyword");

if (defined ($keyword) && $keyword !~ /^\s*$/)
{
  my $dbh = Cookbook::connect ();
  my $found = 0;
  my $s;

  # first look for keyword as a state abbreviation;
  # if that fails, look for it as a name
  $s = $dbh->selectrow_array ("SELECT name FROM states WHERE abbrev = ?",
                              undef, $keyword);
```

```
  if ($s)
  {
    ++$found;
    print p ("You entered the abbreviation: " . escapeHTML ($keyword));
    print p ("The corresponding state name is : " . escapeHTML ($s));
  }
  $s = $dbh->selectrow_array ("SELECT abbrev FROM states WHERE name = ?",
                              undef, $keyword);
  if ($s)
  {
    ++$found;
    print p ("You entered the state name: " . escapeHTML ($keyword));
    print p ("The corresponding abbreviation is : " . escapeHTML ($s));
  }
  if (!$found)
  {
    print p ("You entered the keyword: " . escapeHTML ($keyword));
    print p ("No match was found.");
  }

  $dbh->disconnect ();
}

print p (qq{
Enter a state name into the form and select Search, and I will
show you the corresponding abbreviation. Or enter an abbreviation
and I will show you the full name.
});

print start_form (-action => url ()),
      "State: ",
      textfield (-name => "keyword", -size => 20),
      br (),
      submit (-name => "choice", -value => "Search"),
      end_form ();

print end_html ();
```

The script first checks whether a `keyword` parameter is present. If so, it executes the statements that look for a match to the parameter value in the `states` table and displays the results. Then it presents the form so that the user can enter a new search.

When you try the script, you'll notice that the value of the `keyword` field carries over from one invocation to the next. That's due to CGI.pm's behavior of initializing form fields with values from the script environment. If you don't like this behavior, defeat it and make the field come up blank each time by supplying an empty value explicitly and an `override` parameter in the `textfield()` call:

```
print textfield (-name => "keyword",
                 -value => "",
                 -override => 1,
                 -size => 20);
```

Alternatively, clear the parameter's value in the environment before generating the field:

```
param (-name => "keyword", -value => "");
print textfield (-name => "keyword", -size => 20);
```

20.10. Generating Previous-Page and Next-Page Links

Problem

A statement matches so many rows that displaying them all in a single web page produces an unwieldy result.

Solution

Split the statement output across several pages and include links that enable the user to navigate among pages.

Discussion

If a statement matches a large number of rows, showing them all in a single web page can result in a display that's difficult to navigate. For such cases, it can be more convenient for the user if you split the result among multiple pages. A paged display avoids overwhelming the user with too much information, but is more difficult to implement than a single-page display.

A paged display typically is used in a search context to present rows that match the search parameters supplied by the user. To simplify things, the examples in this recipe don't have any search interface. Instead, they implement a paged display that presents 10 rows at a time from the result of a fixed statement:

```
SELECT name, abbrev, statehood, pop FROM states ORDER BY name;
```

MySQL makes it easy to select just a portion of a result set: add a `LIMIT` clause that indicates which rows you want. The two-argument form of `LIMIT` takes values indicating how many rows to skip at the beginning of the result set, and how many to select. The statement to select a section of the `states` table thus becomes:

```
SELECT name, abbrev, statehood, pop FROM states ORDER BY name
LIMIT skip,select;
```

One issue, then, is to determine the proper values of *`skip`* and *`select`* for any given page. Another is to generate the links that point to other pages or the statement result. This second issue presents you with a choice: which paging style should you use for the links?

- One style of paged display presents only "previous page" and "next page" links. To do this, you must know whether any rows precede or follow those you display in the current page.
- Another paging style displays a link for each available page. This enables the user to jump directly to any page, not just the previous or next page. To present this kind of navigation, you must know the total number of rows in the result set and the number of rows per page, so that you can determine how many pages there are.

Paged displays with previous-page and next-page links

The following script, *state_pager1.pl*, presents rows from the `states` table in a paged display that includes navigation links only to the previous and next pages. For a given page, determine the required links as follows:

- A "previous page" link is required if there are rows in the result set preceding those shown in the current page. If the current page starts at row one, there are no such rows.
- A "next page" link is required if there are rows in the result set following those shown in the current page. You can determine this by issuing a `SELECT COUNT(*)` statement to see how many rows the statement matches in total. Another method is to select one more row than you need. For example, if you display 10 rows at a time, try to select 11 rows. If you get 11, there is a next page. If you get 10 or less, there isn't. *state_pager1.pl* uses the latter approach.

To determine its current position in the result set and how many rows to display, *state_pager1.pl* looks for `start` and `per_page` input parameters. When you first invoke the script, these parameters aren't present, so they're initialized to 1 and 10, respectively. Thereafter, the script generates "previous page" and "next page" links to itself that include the proper parameter values in the URLs for selecting the previous or next sections of the result set:

```
#!/usr/bin/perl
# state_pager1.pl: paged display of states, with prev-page/next-page links

use strict;
use warnings;
use CGI qw(:standard escape escapeHTML);
use Cookbook;

my $title = "Paged U.S. State List";

my $page = header ()
           . start_html (-title => $title)
           . h3 ($title);
```

```
my $dbh = Cookbook::connect ();

# Collect parameters that determine where we are in the display and
# verify that they are integers.
# Default to beginning of result set, 10 records/page if parameters
# are missing/malformed.

my $start = param ("start");
$start = 1
  if !defined ($start) || $start !~ /^\d+$/ || $start < 1;

my $per_page = param ("per_page");
$per_page = 10
  if !defined ($per_page) || $per_page !~ /^\d+$/ || $per_page < 1;;

# If start > 1, then we'll need a live "previous page" link.
# To determine whether there is a next page, try to select one more
# record than we need.  If we get that many, display only the first
# $per_page records, but add a live "next page" link.

# Select the records in the current page of the result set, and
# attempt to get an extra record.  (If we get the extra one, we
# won't display it, but its presence tells us there is a next
# page.)

my $stmt = sprintf ("SELECT name, abbrev, statehood, pop
                     FROM states
                     ORDER BY name LIMIT %d,%d",
                    $start - 1,       # number of records to skip
                    $per_page + 1);   # number of records to select

my $tbl_ref = $dbh->selectall_arrayref ($stmt);

$dbh->disconnect ();

# Display results as HTML table
my @rows;
push (@rows, Tr (th (["Name", "Abbreviation", "Statehood", "Population"])));
for (my $i = 0; $i < $per_page && $i < @{$tbl_ref}; $i++)
{
  # get data values in row $i
  my @cells = @{$tbl_ref->[$i]};  # get data values in row $i
  # map values to HTML-encoded values, or to   if null/empty
  @cells = map {
             defined ($_) && $_ ne "" ? escapeHTML ($_) : " "
           } @cells;
  # add cells to table
  push (@rows, Tr (td (\@cells)));
}

$page .= table ({-border => 1}, @rows) . br ();
```

```
# If we're not at the beginning of the query result, present a live
# link to the previous page.  Otherwise, present static text.

if ($start > 1)         # live link
{
  my $url = sprintf ("%s?start=%d;per_page=%d",
                     url (),
                     $start - $per_page,
                     $per_page);
  $page .= "[" . a ({-href => $url}, "previous page") . "] ";
}
else                    # static text
{
  $page .= "[previous page]";
}

# If we got the extra record, present a live link to the next page.
# Otherwise, present static text.

if (@{$tbl_ref} > $per_page)  # live link
{
  my $url = sprintf ("%s?start=%d;per_page=%d",
                     url (),
                     $start + $per_page,
                     $per_page);
  $page .= "[" . a ({-href => $url}, "next page") . "]";
}
else                          # static text
{
  $page .= "[next page]";
}

$page .= end_html ();

print $page;
```

Paged displays with links to each page

The next script, *state_pager2.pl*, is much like *state_pager1.pl*, but presents a paged display that includes navigation links to each page of the query result. To do this, it's necessary to know how many rows there are in all. *state_pager2.pl* determines this by running a `SELECT COUNT(*)` statement. Because the script then knows the total row count, it need not select an extra row when fetching the section of the result to be displayed. (For a large table, `SELECT COUNT(*)` with no `WHERE` clause can be slow. For an application using a such a table, it's best to include a `WHERE` clause that narrows down the result.)

Omitting the parts of *state_pager2.pl* that are the same as *state_pager1.pl*, the middle part that retrieves rows and generates links is implemented as follows:

```
# Determine total number of records
```

```
my $total_recs = $dbh->selectrow_array ("SELECT COUNT(*) FROM states");

# Select the records in the current page of the result set

my $stmt = sprintf ("SELECT name, abbrev, statehood, pop
                     FROM states
                     ORDER BY name LIMIT %d,%d",
                    $start - 1,     # number of records to skip
                    $per_page);     # number of records to select

my $tbl_ref = $dbh->selectall_arrayref ($stmt);

$dbh->disconnect ();

# Display results as HTML table
my @rows;
push (@rows, Tr (th (["Name", "Abbreviation", "Statehood", "Population"])));
for (my $i = 0; $i < @{$tbl_ref}; $i++)
{
  # get data values in row $i
  my @cells = @{$tbl_ref->[$i]};  # get data values in row $i
  # map values to HTML-encoded values, or to   if null/empty
  @cells = map {
             defined ($_) && $_ ne "" ? escapeHTML ($_) : " "
           } @cells;
  # add cells to table
  push (@rows, Tr (td (\@cells)));
}

$page .= table ({-border => 1}, @rows) . br ();

# Generate links to all pages of the result set.  All links are
# live, except the one to the current page, which is displayed as
# static text.  Link label format is "[m to n]" where m and n are
# the numbers of the first and last records displayed on the page.

for (my $first = 1; $first <= $total_recs; $first += $per_page)
{
  my $last = $first + $per_page - 1;
  $last = $total_recs if $last > $total_recs;
  my $label = "$first to $last";
  my $link;

  if ($first != $start) # live link
  {
    my $url = sprintf ("%s?start=%d;per_page=%d",
                       url (),
                       $first,
                       $per_page);
    $link = a ({-href => $url}, $label);
  }
  else                  # static text
```

```
  {
    $link = $label;
  }
  $page .= "[$link] ";
}
```

20.11. Generating "Click to Sort" Table Headings

Problem

You want to display a query result in a web page as a table that enables the user to select the column by which to sort the table rows.

Solution

Make each column heading a hyperlink that redisplays the table, sorted by the corresponding column.

Discussion

A web script can determine what action to take by examining its environment to find out what parameters are present and what their values are. In many cases these parameters come from a user, but there's no reason a script cannot add parameters to URLs itself. This is one way a given invocation of a script can send information to the next invocation. The effect is that the script communicates with itself by means of URLs that it generates to cause specific actions. An application of this technique is for showing the result of a query such that a user can select which column of the result to use for sorting the display. To do this, make the column headers active links that redisplay the table, sorted by the selected column.

The examples here use the `driver_log` table, which has these contents:

```
mysql> SELECT * FROM driver_log;
+--------+-------+------------+-------+
| rec_id | name  | trav_date  | miles |
+--------+-------+------------+-------+
|      1 | Ben   | 2014-07-30 |   152 |
|      2 | Suzi  | 2014-07-29 |   391 |
|      3 | Henry | 2014-07-29 |   300 |
|      4 | Henry | 2014-07-27 |    96 |
|      5 | Ben   | 2014-07-29 |   131 |
|      6 | Henry | 2014-07-26 |   115 |
|      7 | Suzi  | 2014-08-02 |   502 |
|      8 | Henry | 2014-08-01 |   197 |
|      9 | Ben   | 2014-08-02 |    79 |
|     10 | Henry | 2014-07-30 |   203 |
+--------+-------+------------+-------+
```

To retrieve the table and display its contents as an HTML table, use the techniques discussed in Recipe 19.3. Here we'll use those same concepts but modify them to produce "click to sort" table column headings.

A "plain" HTML table includes a row of column headers consisting only of the column names:

```
<tr>
  <th>rec_id</th>
  <th>name</th>
  <th>trav_date</th>
  <th>miles</th>
</tr>
```

To make the headings active links that reinvoke the script to produce a display sorted by a given column name, we must produce a header row that looks like this:

```
<tr>
  <th><a href="script_name?sort=rec_id">rec_id</a></th>
  <th><a href="script_name?sort=name">name</a></th>
  <th><a href="script_name?sort=trav_date">trav_date</a></th>
  <th><a href="script_name?sort=miles">miles</a></th>
</tr>
```

To generate such headings, the script must know the names of the columns in the table, as well as its own URL. Recipes 10.6 and 20.1 show how to obtain this information using statement metadata and information from the script's environment. For example, in PHP, a script can generate the header row for the columns in a given statement as follows, where `getColumnMeta(`*`i`*`)` returns metadata for column *`i`*:

```
print ('<tr>');
$ncols = $sth->columnCount ();
for ($i = 0; $i < $ncols; $i++)
{
  $col_info = $sth->getColumnMeta ($i);
  $col_name = $col_info['name'];
  printf ('<th><a href="%s?sort=%s">%s</a></th>',
          $_SERVER['PHP_SELF'],
          urlencode ($col_name),
          htmlspecialchars ($col_name));
}
print ('</tr>');
```

The following script, *clicksort.php*, implements this kind of table display. It checks its environment for a `sort` parameter that indicates which column to use for sorting, then uses the parameter to construct a statement of the following form:

```
SELECT * FROM $tbl_name ORDER BY $sort_col LIMIT 50
```

There is a small bootstrapping problem for this kind of script. The first time you invoke it, there is no sort column name in the environment, so the script doesn't know which column to sort by initially. What should you do? There are several possibilities:

- Retrieve the results unsorted.
- Hardwire one of the column names into the script as the default.
- Retrieve the column names from `INFORMATION_SCHEMA` and use one of them (such as the first) as the default:

```
SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = "cookbook" AND TABLE_NAME = "mail"
AND ORDINAL_POSITION = 1;
```

The following script looks up the name from `INFORMATION_SCHEMA`. It also uses a `LIMIT` clause when retrieving results as a precaution that prevents the script from dumping huge amounts of output if the table is large:

```
<?php
# clicksort.php: display query result as HTML table with "click to sort"
# column headings

# Rows from the database table are displayed as an HTML table.
# Column headings are presented as hyperlinks that reinvoke the
# script to redisplay the table sorted by the corresponding column.
# The display is limited to 50 rows in case the table is large.

require_once "Cookbook.php";
require_once "Cookbook_Utils.php";
require_once "Cookbook_Webutils.php";

$title = "Table Display with Click-To-Sort Column Headings";
?>

<html>
<head><title><?php print ($title); ?></title></head>
<body>

<?php
# names for database and table and default sort column; change as desired
$db_name = "cookbook";
$tbl_name = "driver_log";

$dbh = Cookbook::connect ();

print ("<p>" . htmlspecialchars ("Table: $db_name.$tbl_name") . "</p>");
print ("<p>Click a column name to sort by that column.</p>");

# Get the name of the column to sort by: If missing, use the first column.

$sort_col = get_param_val ("sort");
if (!isset ($sort_col))
{
  $stmt = "SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS
           WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?
```

```
          AND ORDINAL_POSITION = 1";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ($db_name, $tbl_name));
  list ($sort_col) = $sth->fetch (PDO::FETCH_NUM);
}

# Construct query to select records from the table, sorting by the
# named column. Limit output to 50 rows to avoid dumping entire
# contents of large tables.

$stmt = sprintf ("SELECT * FROM %s.%s ORDER BY %s LIMIT 50",
                 quote_identifier ($db_name),
                 quote_identifier ($tbl_name),
                 quote_identifier ($sort_col));
$sth = $dbh->query ($stmt);

# Display query results as HTML table.  Use query metadata to get column
# names, and display names in first row of table as hyperlinks that cause
# the table to be redisplayed, sorted by the corresponding table column.

print ('<table border="1">');
print ('<tr>');
$ncols = $sth->columnCount ();
for ($i = 0; $i < $ncols; $i++)
{
  $col_info = $sth->getColumnMeta ($i);
  $col_name = $col_info['name'];
  printf ('<th><a href="%s?sort=%s">%s</a></th>',
          $_SERVER['PHP_SELF'],
          urlencode ($col_name),
          htmlspecialchars ($col_name));
}
print ('</tr>');
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  print ('<tr>');
  for ($i = 0; $i < $ncols; $i++)
  {
    # encode values, using   for empty cells
    $val = $row[$i];
    if (isset ($val) && $val != '')
      $val = htmlspecialchars ($val);
    else
      $val = ' ';
    printf ('<td>%s</td>', $val);
  }
  print ('</tr>');
}
print ('</table>');

$dbh = NULL;
?>
```

```
</body>
</html>
```

The `$sort_col` value comes from the `sort` parameter of the environment, so it should be considered dangerous: An attacker might submit a URL with a `sort` parameter designed to attempt an SQL injection attack. To prevent this, `$sort_col` should be quoted when you construct the `SELECT` statement that retrieves rows from the displayed table. You cannot use a placeholder to quote the value because that technique applies to data values. (`$sort_col` is an identifier here, not a data value.) *clicksort.php* uses the `quote_identifier()` function from *Cookbook_Utils.php* to make the identifiers safe for inclusion in the SQL statement (see Recipe 2.6).

Another approach to validating the column name is to check the `COLUMNS` table of `INFORMATION_SCHEMA`. This enables you to incorporate the table name into the query as a data value, so it can be supplied using a placeholder. If the sort column is not found, it is invalid. The *clicksort.php* script shown here does not do that. However, the `recipes` distribution contains a Perl counterpart script, *clicksort.pl*, that does perform this kind of check. Have a look at it if you want more information.

The cells in the rows following the header row contain the data values from the database table, displayed as static text. Empty cells are displayed using ` ` so that they display with the same border as nonempty cells (see Recipe 19.3).

20.12. Web Page Access Counting

Problem

You want to count the number of times web pages have been accessed.

Solution

Implement a hit counter, keyed to the page to be counted. This can be used to display a counter in the page. Use the same technique to record other types of information as well, such as the number of times each of a set of banner ads has been served.

Discussion

This recipe discusses access counting, using hit counters for the examples. Counters that display the number of times a web page has been accessed are not such a big thing as they used to be, presumably because page authors now realize that they serve primarily the author's vanity; most visitors don't really care how popular a page is. Still, the general concept has application in many contexts. For example, if you display banner ads in your pages (see Recipe 19.7), you likely charge vendors by the number of times

you serve their ads. To do so, you must count the number of accesses for each. The technique shown in this section can be adapted for all such purposes.

There are several methods for writing a page that displays a count of the number of times it has been accessed. The most basic is to maintain the count in a file. For each page request, open the file, read the count, increment it, write the new count to the file, and display it in the page. This is easy to implement but requires a counter file for each page that includes a hit count. It also doesn't work properly if two clients access the page at the same time, unless you implement some kind of locking protocol in the file-access procedure. It's possible to reduce counter file litter by keeping multiple counts in a single file, but that makes it more difficult to access particular values within the file, and it doesn't solve the simultaneous-access problem. In fact, the problem is worse because a multiple-counter file has a higher likelihood of being accessed by multiple clients simultaneously than does a single-counter file. So you end up implementing storage and retrieval methods for processing the file contents, and locking protocols to keep multiple processes from interfering with each other. Hmm... those sound suspiciously like the problems that a database management system such as MySQL already takes care of! Keeping the counts in the database centralizes them into a single table, SQL provides the storage and retrieval interface, and the locking problem goes away because MySQL serializes access to the table so that clients can't interfere with each other. Furthermore, depending on how you manage the counters, you might be able to update the counter and retrieve the new sequence value using a single statement.

Assume that you want to log hits for more than one page. To do that, create a table that has one row for each page to be counted. This necessitates a unique identifier for each page, so that counters for different pages don't get mixed up. Each page's path within your web tree is unique, so just use that. (Web programming languages typically make this path easy to obtain, as discussed in Recipe 20.1.) On that basis, create a `hitcount` table as follows:

```
CREATE TABLE hitcount
(
  path  VARCHAR(255)
          CHARACTER SET latin1 COLLATE latin1_general_cs NOT NULL,
  hits  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (path)
);
```

This table definition involves some assumptions:

- The `path` column that stores page pathnames has a character set of `latin1` and a case-sensitive collation of `latin1_general_cs`. Use of a case-sensitive collation is appropriate for a web platform where pathnames are case sensitive, such as most versions of Unix. For Windows or for HFS+ filesystems under Mac OS X, filenames are not case sensitive, so you would choose a collation that is not case sensitive,

such as `latin1_swedish_ci`. If your filesystem is set up to use pathnames in a different character set, change the character set and collation.

- The `path` column has a maximum length of 255 characters, which limits you to page paths no longer than that.
- The `path` column is indexed as a `PRIMARY KEY` to require unique values. Either a `PRIMARY KEY` or `UNIQUE` index is required because we will implement the hit-counting algorithm using an `INSERT` statement with an `ON DUPLICATE KEY UPDATE` clause to insert a row if none exists for the page or update the row if it does exist. (Recipe 13.12 explains `ON DUPLICATE KEY UPDATE`.)
- The table is set up to count page hits for a single document tree, such as when your web server is used to serve pages for a single domain. If you institute a hit count mechanism on a host that serves multiple virtual domains, you may want to add a column for the domain name. This value is available in the `SERVER_NAME` value that Apache puts into your script's environment. In this case, the `hitcount` table index should include both the hostname and the page path.

The general logic involved in hit counter maintenance is to increment the `hits` column of the row for a page, and then retrieve the updated counter value:

```
UPDATE hitcount SET hits = hits + 1 WHERE path = 'page path';
SELECT hits FROM hitcount WHERE path = 'page path';
```

Unfortunately, with that approach, you might not get the correct value. If several clients request the page simultaneously, several `UPDATE` statements execute in close temporal proximity and the `SELECT` statements that follow won't necessarily get the corresponding `hits` value. This can be avoided by using a transaction or by locking the `hitcount` table, but that slows down hit counting. MySQL provides a solution that enables each client to retrieve its own count, no matter how many simultaneous updates occur:

```
UPDATE hitcount SET hits = LAST_INSERT_ID(hits+1) WHERE path = 'page path';
SELECT LAST_INSERT_ID();
```

The basis for updating the count here is `LAST_INSERT_ID(`*`expr`*`)`, discussed in Recipe 13.12. The `UPDATE` statement finds the relevant row and increments its counter value. The use of `LAST_INSERT_ID(hits+1)` rather than just `hits+1` tells MySQL to treat the value as though it were an `AUTO_INCREMENT` value. This enables it to be retrieved in the second statement using `LAST_INSERT_ID()`. The `LAST_INSERT_ID()` function returns a connection-specific value, so it always corresponds to the preceding `UPDATE` for the same connection. In addition, the `SELECT` statement doesn't query a table, so it's very fast.

However, there's still a problem. What if the page isn't listed in the `hitcount` table? In that case, the `UPDATE` statement finds no row to modify and you get a counter value of zero. You could deal with this problem by requiring that any page that includes a hit counter must be registered in the `hitcount` table before the page goes online. This is

tedious and difficult to enforce. An easier approach is to use MySQL's `INSERT ... ON DUPLICATE KEY UPDATE` syntax to insert a row with a count of 1 if it does not exist, and update its counter if it does exist:

```
INSERT INTO hitcount (path,hits) VALUES('some path',LAST_INSERT_ID(1))
ON DUPLICATE KEY UPDATE hits = LAST_INSERT_ID(hits+1);
SELECT LAST_INSERT_ID();
```

The first time you request a count for a page, the statement inserts a row because the page isn't listed in the table yet. The statement creates a new counter and initializes it to one. For each request thereafter, the statement updates the existing row for the page with the new count. No advance page registration in the `hitcount` table is required.

If your API provides a means for direct retrieval of the most recent sequence number, a further efficiency is gained by eliminating the `SELECT` statement altogether. For example, in Perl, you can update the count and get the new value with only one SQL statement like this:

```
$dbh->do ("INSERT INTO hitcount (path,hits) VALUES(?,LAST_INSERT_ID(1))
           ON DUPLICATE KEY UPDATE hits = LAST_INSERT_ID(hits+1)",
          undef, $page_path);
$count = $dbh->{mysql_insertid};
```

To make the counter mechanism easier to use, put the code in a utility function that takes a page path as an argument and returns the count. In Perl, a hit-counting function might look like this, in which the arguments are a database handle and the page path:

```
sub get_hit_count
{
my ($dbh, $page_path) = @_;

  $dbh->do ("INSERT INTO hitcount (path,hits) VALUES(?,LAST_INSERT_ID(1))
             ON DUPLICATE KEY UPDATE hits = LAST_INSERT_ID(hits+1)",
            undef, $page_path);
  return $dbh->{mysql_insertid};
}
```

The CGI.pm `script_name()` function returns the local part of the URL, so use `get_hit_count()` like this:

```
my $count = get_hit_count ($dbh, script_name ());
print p ("This page has been accessed $count times.");
```

The technique is analogous for other languages. For example, the Ruby version of the hit counter looks like this:

```
def get_hit_count(dbh, page_path)
  dbh.do("INSERT INTO hitcount (path,hits) VALUES(?,LAST_INSERT_ID(1))
          ON DUPLICATE KEY UPDATE hits = LAST_INSERT_ID(hits+1)",
         page_path)
  return dbh.func(:insert_id)
end
```

Use the counter method as follows:

```
count = get_hit_count(dbh, ENV["SCRIPT_NAME"])
page << cgi.p { "This page has been accessed #{count} times." }
```

The `recipes` distribution includes demonstration hit counter scripts in the *apache/hits* directory (*tomcat/mcb* for JSP). Install any of these in your web tree, invoke it from your browser a few times, and watch the count increase. First, you must create the `hitcount` table. To do this, use the *hits.sql* script provided in the *tables* directory. (The script also creates the `hitlog` table because the hit-counting scripts implement hit logging as well, as discussed in Recipe 20.13.)

20.13. Web Page Access Logging

Problem

You want to know things about a page other than the number of times it's been accessed, such as when accesses occur and the hosts from which requests originate.

Solution

Maintain a hit log rather than a simple counter.

Discussion

The `hitcount` table used in Recipe 20.12 records only the access count for each page registered in it. Suppose that you want to track other information about page access, such as the time of access and client host for each request. In this case, you must log a row for each page access rather than maintain only a count:

```
CREATE TABLE hitlog
(
  path  VARCHAR(255)
          CHARACTER SET latin1 COLLATE latin1_general_cs NOT NULL,
  t     TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  host  VARCHAR(255),
  INDEX (path)
);
```

The `hitlog` table has the useful property that access times are recorded automatically in the `TIMESTAMP` column `t` when you insert new rows (see Recipe 6.7). For notes on choosing the character set and collation for the `path` column, see Recipe 20.12.

To insert new rows into the `hitlog` table, use this statement:

```
INSERT INTO hitlog (path, host) VALUES(path_val,host_val);
```

For example, in a JSP page, log hits like this:

```
<c:set var="host"><%= request.getRemoteHost () %></c:set>
<c:if test="${empty host}">
  <c:set var="host"><%= request.getRemoteAddr () %></c:set>
</c:if>
<c:if test="${empty host}">
  <c:set var="host">UNKNOWN</c:set>
</c:if>

<sql:update dataSource="${conn}">
  INSERT INTO hitlog (path, host) VALUES(?,?)
  <sql:param><%= request.getRequestURI () %></sql:param>
  <sql:param value="${host}"/>
</sql:update>
```

Although the `hitlog` table doesn't maintain page-access counts, you can determine them easily:

- To determine the number of hits for a given page, use this statement:

  ```
  SELECT COUNT(*) FROM hitlog WHERE path = 'path_name';
  ```

- To determine the current counter value for all pages and retrieve them in order with the most-requested pages first, do this:

  ```
  SELECT path, COUNT(*) FROM hitlog GROUP BY path ORDER BY COUNT(*) DESC;
  ```

20.14. Using MySQL for Apache Logging

Problem

You don't want to use MySQL to log accesses for just a few pages, as shown in Recipe 20.13. You want to log all page accesses, without having to put explicit logging code in each page.

Solution

Tell Apache to log page accesses by writing to a MySQL table.

Discussion

The uses for MySQL in a web context aren't limited to page generation and processing. MySQL can help you run the web server itself. For example, most Apache servers are set up to log a record of page requests to a file. But it's also possible to send log records to a program instead, from which you can write the records wherever you like—such as to a database. Logging records in a database rather than a flat file makes the log more highly structured and you can apply SQL analysis techniques to it. Want to see a particular report? Write the SQL statements that produce it. To display the report in a

specific format, issue the statements from within an API and take advantage of your language's output production capabilities.

By handling log entry generation and storage using separate processes, you gain flexibility. Some of the possibilities are to send logs from multiple web servers to the same MySQL server, or to send different logs generated by a given web server to different MySQL servers.

This recipe shows how to integrate MySQL into Apache's logging mechanism and demonstrates some representative summary queries.

Setting up database logging

Directives in the *httpd.conf* configuration file control Apache logging. For example, a typical logging setup uses `LogFormat` and `CustomLog` directives that look like this:

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog /usr/local/apache/logs/access_log common
```

The `LogFormat` line defines a format for log records and gives it the nickname `common`. The `CustomLog` directive indicates that lines should be written in that format to the *access_log* file in Apache's *logs* directory. To set up logging to MySQL instead, use the following procedure. (Adapt it as necessary if you use logging directives such as `TransferLog` rather than `LogFormat` and `CustomLog`.)

1. Decide what values to record and set up a table that contains the appropriate columns.
2. Write a program that reads log lines from Apache and writes them to the database.
3. Set up a `LogFormat` line that defines how to write log lines in the format the program expects, and a `CustomLog` directive that tells Apache to write to the program rather than to a file.

Suppose that you want to record the date and time of each request, the host that issued the request, the request method and URL pathname, the status code, the number of bytes transferred, the referring page, and the user agent (typically a browser or spider name). The following table includes columns for these values:

```
CREATE TABLE httpdlog
(
  dt      DATETIME NOT NULL,            # request date
  host    VARCHAR(255) NOT NULL,        # client host
  method  VARCHAR(4) NOT NULL,          # request method (GET, PUT, etc.)
  url     VARCHAR(255)                  # URL path
            CHARACTER SET latin1 COLLATE latin1_general_cs NOT NULL,
  status  INT NOT NULL,                 # request status
  size    INT,                          # number of bytes transferred
  referer VARCHAR(255),                 # referring page
```

```
  agent   VARCHAR(255)                # user agent
);
```

Most of the string columns use `VARCHAR` and are not case sensitive. The exception, `url`, is declared with a case-sensitive collation as is appropriate for a server running on a system with case-sensitive filenames. For notes on choosing the character set and collation for the `path` column, see Recipe 20.12.

The `httpdlog` table definition shown here includes no indexes. If you plan to run summary queries, add appropriate indexes to the table. Otherwise, the summaries slow dramatically as table size increases. The columns to index depend on the types of statements you intend to run to analyze the table contents. For example, statements that analyze the distribution of client host values benefit from an index on the `host` column.

Next, you need a program to process log lines produced by Apache and insert them into the `httpdlog` table. The following script, *httpdlog.pl*, opens a connection to the MySQL server, then loops to read input lines. It parses each line into column values and inserts the result into the database. When Apache exits, it closes the pipe to the logging program. *httpdlog.pl* sees end of file on its input, terminates the loop, and disconnects from MySQL:

```
#!/usr/bin/perl
# httpdlog.pl: Log Apache requests to httpdlog table

# path to directory containing Cookbook.pm (*** CHANGE AS NECESSARY ***)
use lib qw(/usr/local/lib/mcb);
use strict;
use warnings;
use Cookbook;

my $dbh = Cookbook::connect ();
my $sth = $dbh->prepare (qq{
            INSERT INTO httpdlog
              (dt,host,method,url,status,size,referer,agent)
              VALUES (?,?,?,?,?,?,?,?)
          });

while (<>)  # loop while there is input to read
{
  chomp;
  my ($dt, $host, $method, $url, $status, $size, $refer, $agent)
    = split (/\t/, $_);
  # map "-" to NULL for some columns
  $size = undef if $size eq "-";
  $agent = undef if $agent eq "-";
  $sth->execute ($dt, $host, $method, $url,
                 $status, $size, $refer, $agent);
}

$dbh->disconnect ();
```

The purpose of including the `use lib` line is so that Perl can find the *Cookbook.pm* module. This line is needed if the environment of scripts invoked by Apache for logging does not enable Perl to find the module. Adjust the path as necessary for your system. Alternatively, modify the Apache environment using a `SetEnv` directive (see Recipe 18.2).

httpdlog.pl assumes that input lines contain `httpdlog` column values delimited by tabs (to make it easy to break input lines), so Apache must write log entries in a matching format. The following table shows the `LogFormat` field specifiers to produce the appropriate values:

Specifier	Meaning
`%{%Y-%m-%d %H:%M:%S}t`	The date and time of the request, in MySQL's `DATETIME` format
`%h`	The host from which the request originated
`%m`	The request method (`get`, `post`, and so forth)
`%U`	The URL path
`%>s`	The status code
`%b`	The number of bytes transferred
`%{Referer}i`	The referring page
`%{User-Agent}i`	The user agent

To define a logging format named `mysql` that produces these values with tabs in between, add the following `LogFormat` directive to your *httpd.conf* file:

```
LogFormat \
"%{%Y-%m-%d %H:%M:%S}t\t%h\t%m\t%U\t%>s\t%b\t%{Referer}i\t%{User-Agent}i" \
mysql
```

Most of the pieces are in place now. We have a log table, a program that writes to it, and a `mysql` format for producing log entries. Install the *httpdlog.pl* script where you want Apache to look for it. On my system, the Apache root directory is */usr/local/apache*, so */usr/local/apache/bin* is a reasonable installation directory. This path is needed shortly for constructing the `CustomLog` directive that instructs Apache to log to the script.

All that remains is to tell Apache to write the entries to the *httpdlog.pl* script. However, until you know that the output format really is correct and that the program can process log entries properly, it's premature to tell Apache to log directly to the program. To make testing and debugging a bit easier, have Apache log `mysql`-format entries to a file instead. That way, you can look at the file to check the output format, and you can use it as input to *httpdlog.pl* to verify that the program works correctly. To instruct Apache to log lines in `mysql` format to the file *test_log* in Apache's log directory, use this `CustomLog` directive:

```
CustomLog /usr/local/apache/logs/test_log mysql
```

Then restart Apache to enable the new logging directives. After your web server receives a few requests, take a look at the *test_log* file. Verify that the contents are as you expect, then feed the file to *httpdlog.pl*:

```
% /usr/local/apache/bin/httpdlog.pl test_log
```

After *httpdlog.pl* finishes, take a look at the `httpdlog` table to make sure that it looks correct. Once you're satisfied, tell Apache to send log entries directly to *httpdlog.pl* by modifying the `CustomLog` directive as follows:

```
CustomLog "|/usr/local/apache/bin/httpdlog.pl" mysql
```

The `|` character at the beginning of the pathname tells Apache that *httpdlog.pl* is a program, not a file. Restart Apache and new entries should appear in the `httpdlog` table as visitors request pages from your site.

Nothing you have done to this point changes any logging you may have been doing originally. For example, if you were logging to an *access_log* file before, you still are now, so Apache is sending entries both to the original logfile and to MySQL. If that's what you want, fine. Apache doesn't care if you log to multiple destinations, but you'll use more disk space. To disable file logging, disable your original `CustomLog` directive and restart Apache.

Analyzing the logfile

Now that Apache is logging into the database, what you do with the information depends on what you want to know. Here are some questions MySQL can answer easily:

- How many requests were received?

```
SELECT COUNT(*) FROM httpdlog;
```

- How many different client hosts sent requests?

```
SELECT COUNT(DISTINCT host) FROM httpdlog;
```

- How many different pages did clients request?

```
SELECT COUNT(DISTINCT url) FROM httpdlog;
```

- What are the 10 most popular pages?

```
SELECT url, COUNT(*) AS count FROM httpdlog
GROUP BY url ORDER BY count DESC LIMIT 10;
```

- How many requests were received for *favicon.ico* files that certain browsers like to check for?

```
SELECT COUNT(*) FROM httpdlog WHERE url LIKE '%/favicon.ico%';
```

- What is the range of dates spanned by the log?

```
SELECT MIN(dt), MAX(dt) FROM httpdlog;
```

- How many requests were received each day?

```
SELECT DATE(dt) AS day, COUNT(*) FROM httpdlog GROUP BY day;
```

Answering this question requires stripping the time-of-day part from the `dt` values so that requests received on a given date can be grouped. The statement does this using the `DATE()` function to convert `DATETIME` values to `DATE` values. However, if you intend to run a lot of statements that use just the date part of the `dt` values, it would be more efficient to create the `httpdlog` table with separate `DATE` and `TIME` columns, change the `LogFormat` directive to produce the date and time as separate output values, and modify *httpdlog.pl* accordingly. Then you can operate on the request dates directly without stripping the time, and you can index the date column for even better performance.

- What is the hour-of-the-day request histogram?

```
SELECT HOUR(dt) AS hour, COUNT(*) FROM httpdlog GROUP BY hour;
```

- What is the average number of requests received each day?

```
SELECT COUNT(*)/(DATEDIFF(MAX(dt), MIN(dt)) + 1) FROM httpdlog;
```

The numerator is the number of requests in the table. The denominator is the number of days spanned by the records.

- What is the longest URL recorded in the table?

```
SELECT MAX(LENGTH(url)) FROM httpdlog;
```

If the `url` column is defined as `VARCHAR(255)` and this statement produces a value of 255, it's likely that some URL values were too long to fit in the column and were truncated at the end. To avoid this, change the column definition to permit more characters. For example, to permit up to 5,000 characters, modify the `url` column as follows:

```
ALTER TABLE httpdlog
  MODIFY url VARCHAR(5000)
  CHARACTER SET latin1 COLLATE latin1_general_cs NOT NULL;
```

- What is the total number of bytes served and the average bytes per request?

```
SELECT
  COUNT(size) AS requests,
  SUM(size) AS bytes,
  AVG(size) AS 'bytes/request'
FROM httpdlog;
```

The statement uses `COUNT(size)` rather than `COUNT(*)` to count only those requests with a non-`NULL` `size` value. If a client requests a page twice, the server may respond to the second request by sending a header indicating that the page hasn't changed rather than by sending content. In this case, the log entry for the request will have `NULL` in the `size` column.

- How much traffic has there been for each kind of file (based on filename extension such as *.html*, *.jpg*, or *.php*)?

```
SELECT
  SUBSTRING_INDEX(SUBSTRING_INDEX(url,'?',1),'.',-1) AS extension,
  COUNT(size) AS requests,
  SUM(size) AS bytes,
  AVG(size) AS 'bytes/request'
FROM httpdlog
WHERE url LIKE '%.%'
GROUP BY extension;
```

The `WHERE` clause selects only `url` values that have a period in them, to eliminate pathnames that refer to files that have no extension. To extract the extension values for the output column list, the inner `SUBSTRING_INDEX()` call strips any parameter string from the right end of the URL and leaves the rest. (This turns a value like `/cgi-bin/script.pl?id=43` into `/cgi-bin/script.pl`. If the value has no parameter part, `SUBSTRING_INDEX()` returns the entire string.) The outer `SUBSTRING_INDEX()` call strips everything up to and including the rightmost period from the result, leaving only the extension.

Other logging issues

The preceding discussion shows a simple method for hooking Apache to MySQL: write a short script that communicates with MySQL, and tell Apache to write to the script rather than to a file. This works well if you log all requests to a single file, but certainly isn't appropriate for every possible configuration of which Apache is capable. For example, if you have virtual servers defined in your *httpd.conf* file, you might have separate `CustomLog` directives defined for each server. To log them all to MySQL, you can change each directive to write to *httpdlog.pl*, but that results in a separate logging process for each virtual server.

That brings up the issue of how you associate log records with the proper virtual server. One solution is to create a separate log table for each server and modify *httpdlog.pl* to take an argument that indicates which table to use. Another is to use a table that has a `vhost` column, an Apache log format that includes the `%v` virtual host format specifier, and a logging script that uses the `vhost` value when it generates `INSERT` statements. The *apache/httpdlog* directory of the `recipes` distribution contains an *httpdlog2.pl* script that implements this method, along with instructions for using it.

Logging to a database rather than a file enables you to bring the full power of MySQL to bear on log analysis, but it doesn't eliminate the need to think about space management. Web servers can generate a lot of activity, and log records use space regardless of whether you write them to a file or to a database. One way to save space is to expire records now and then. For example, to remove log records that are more than a year old, run the following statement periodically:

```
DELETE FROM httpdlog WHERE dt < NOW() - INTERVAL 1 YEAR;
```

To do this automatically, set up a scheduled event that runs the `DELETE` statement periodically (see Recipe 9.8).

With respect to disk space consumed by web logging activity, be aware that if you have query logging enabled for the MySQL server, each request will be written to the `httpdlog` table and also to the query log. Thus, you may find disk space disappearing more quickly than you expect, so it's a good idea to have some kind of logfile rotation or expiration set up. For some ideas, see Recipe 22.4.

CHAPTER 21

Using MySQL-Based Web Session Management

21.0. Introduction

Many web applications interact with users over a series of requests and, as a result, must retain information from one request to the next. A set of related requests is called a session. Sessions are useful for activities such as performing login operations and associating a logged-in user with subsequent requests, and gathering input from a user in stages (possibly using earlier responses to tailor later questions). However, HTTP is a stateless protocol, so web servers treat each request independently of any other—unless you take steps to ensure otherwise.

This chapter shows how to make information persist across multiple requests, which enables you to develop applications for which one request retains memory of previous ones. The techniques shown here are general enough to apply to a variety of state-maintaining web applications.

Session Management Issues

Some session management methods rely on information stored on the client. One way to implement client-side storage is to use cookies, implemented as information transmitted back and forth in special request and response headers. When a session begins, the application generates and sends the client a cookie containing the initial information to be stored. The client returns the cookie to the server with subsequent requests to identify itself and enable the application to recognize the requests as stages of the same client session. At each stage, the application uses the cookie content to determine the state (or status) of the client. To modify the session state, the application sends the client a new cookie containing updated information to replace the old cookie. This mechanism enables data to persist across requests while still affording the application the

opportunity to update the information as necessary. Cookies are easy to use, but it's possible for the client to modify cookie contents, possibly tricking the application into misbehaving. Other client-side session storage techniques suffer the same drawback.

The alternative to client-side storage is to maintain session state on the server side. Information about client activity is stored somewhere on the server, such as in a file, in shared memory, or in a database. The only information maintained on the client side is a unique identifier that the server generates and sends to the client when the session begins. The client sends this value to the server with each subsequent request so the server can associate the client with the appropriate session. Common techniques for tracking the session ID are to store it in a cookie or to encode it in request URLs (useful for clients that have cookies disabled). The server gets the ID from the cookie value or by extracting it from the URL.

Server-side session storage is more secure than storing information on the client because the application maintains control over the session contents. The only value present on the client side is the session ID, so the client can't modify session data unless the application permits it. It's still possible for a client to tinker with the ID and send back a different one, but if IDs are unique and selected from a very large pool of possible values, a malicious client is unlikely to guess the ID of another valid session. If you are concerned about other clients stealing valid session IDs by network snooping, use sessions within the context of secure connections (for example, by using SSL). But that is beyond the scope of this book.

Server-side methods for managing sessions commonly store session contents in persistent backing storage such as a file or a database. Database storage characteristics differ from file storage, such as that you eliminate the filesystem clutter that results from having many session files, and you can use the same MySQL server to handle session traffic for multiple web servers. If this appeals to you, the techniques shown in the chapter enable you to integrate MySQL-based session management into your applications. The chapter shows how to implement server-side database-backed session management for several of our API languages:

- The Perl Apache::Session module includes most of the capabilities needed for session management. It can store session information in files or in any of several database systems, including MySQL, PostgreSQL, and Oracle.
- The Ruby `CGI::Session` class provides session-handling capability. It uses temporary files by default, but permits other storage managers to be used, such as the `mysql-session` package for MySQL.
- PHP includes native session support. The implementation uses temporary files by default, but applications can supply their own handler routines for session storage. This makes it possible to plug in a storage module that uses MySQL.

- For Java-based web applications running under the Tomcat web server, Tomcat provides session support at the server level. Tomcat uses temporary files by default, but you need only modify a server or application configuration file to use MySQL for session storage. There are no changes to application programs, which need do nothing to select one session backing-store method or another.

Session support for different APIs can use very different approaches. For Perl, the language itself provides no session support, so a script must include a module such as Apache::Session explicitly if it wants to implement a session. Ruby is similar. In PHP, the session manager is built in. Scripts can use it with no special preparation, but only as long as they want to use the default storage method (files). To use an alternative method (such as storing sessions in MySQL), an application must register its own routines with the session manager. Still another approach is used for Java applications running under Tomcat, because Tomcat itself manages many of the details associated with session management, including where to store session data. Individual applications need not know or care where this information is stored.

Despite their differences, session management implementations typically perform a common set of tasks:

- Determine whether the client provided a session ID. If not, generate a unique session ID and send it to the client. Some session managers transmit the session ID between the server and the client automatically. PHP does this, as does Tomcat for Java programs. The Perl Apache::Session module leaves it to the application developer to manage ID transmission.
- Store values into the session for use by later requests and retrieve values placed into the session by earlier requests. This involves any activity that uses session data: increment a counter, validate a login request, update a shopping cart, and so forth.
- Terminate the session when it's no longer needed. Some session managers can expire sessions automatically after a period of inactivity. Sessions may also end explicitly, if the request indicates that the session should terminate (such as when the client selects a logout action). In response, the session manager destroys the session record. It might also be necessary to tell the client to release information. If the client sends the session identifier in a cookie, the application should instruct the client to discard the cookie. Otherwise, the client may continue to submit it after it no longer applies. Another approach to session "termination" is to delete all information from the session record. This causes a new session to start with the client's next request because no previous session information is available.

Session managers impose little constraint on what applications can store in session records. Sessions usually can accommodate various types of data, such as scalars, arrays, or objects. To make it easy to store and retrieve session data, session managers typically serialize session information by converting it to a coded scalar string value before storing

it and unserialize it after retrieval. The conversion to and from serialized strings generally is not something you must deal with when providing storage routines. It's necessary only to make sure the storage manager has a large enough repository in which to store the serialized strings. For backing store implemented using MySQL, use one of the `BLOB` data types. Our session managers use the largest such type, `LONGBLOB`. When assessing storage needs, remember that stored data is serialized, which takes more space than raw data.

The rest of the chapter shows a session-based script for each API. Each script performs two tasks. It maintains a counter value that indicates how many requests have been received during the current session, and records a timestamp for each request. In this way, the scripts illustrate how to store and retrieve a scalar value (the counter) and a nonscalar value (the timestamp array). They require very little interaction with the user, who simply reloads the page to issue the next request. This results in extremely simple code.

Session-based applications often include some way for the user to log out explicitly and terminate the session. The example scripts implement a form of "logout" based on an implicit mechanism: sessions have a limit of 10 requests. As you reinvoke a script, it checks whether the counter limit has been reached and destroys the session data if so. Because the session values are not present in the next request, the script starts a new session.

To see the queries that MySQL-based session managers generate, watch the server's general query log as you invoke session scripts from your browser. (The log must be enabled; see Recipe 22.3.)

The example session scripts for Perl, Ruby, and PHP are located under the *apache* directory of the `recipes` distribution; the PHP session module is located in the *lib* directory; and the JSP examples are under the *tomcat* directory. SQL scripts for creating the session storage tables are located in the *tables* directory. As used here, the session tables are created in the `cookbook` database and accessed through the same MySQL account as that used elsewhere in this book. If you don't want to mix session management activities with those pertaining to the other `cookbook` tables, consider setting up a separate database and MySQL account to be used only for session data. This is true particularly for Tomcat, where session management takes place above the application level. You might not want the Tomcat server storing information in "your" database; if not, give Tomcat its own database.

21.1. Using MySQL-Based Sessions in Perl Applications

Problem

You want to use session storage for Perl scripts.

Solution

The Apache::Session module provides a convenient way to use several different storage types, including one based on MySQL.

Discussion

Apache::Session is an easy-to-use Perl module for maintaining state information across multiple web requests. Despite the name, this module does not require Apache and can be used in nonweb contexts—for example, to maintain persistent state across multiple invocations of a command-line script.

Apache::Session does not handle issues associated with tracking the session ID (sending it to the client in response to the initial request and extracting it from subsequent requests). The example application shown here uses cookies to pass the session ID, on the assumption that the client has cookies enabled.

Apache::Session setup

If Apache::Session is not installed, get it from CPAN. Apache::Session also requires several other modules that you may need to install first. (If you use a *cpan install Apache::Session* command, that should install the module and take care of dependencies.) After you have everything installed, create a table in which to store session records, in any database you like (we'll use `cookbook`). The specification for the table comes from the MySQL storage handler documentation, which you can read using this command:

```
% perldoc Apache::Session::Store::MySQL
```

We'll use a table named `perl_session` with this structure:

```
CREATE TABLE perl_session
(
  id        CHAR(32) NOT NULL, # session identifier
  a_session LONGBLOB,          # session data
  PRIMARY KEY (id)
);
```

The `id` column holds session identifiers, which are unique 32-character MD5 values generated by Apache::Session. The `a_session` column stores session data as serialized strings. Apache::Session uses the Storable module to serialize and unserialize session data. (The Apache::Session::Store::MySQL documentation indicates that `a_session` is a `TEXT` column, but any `BLOB` or `TEXT` data type large enough to hold the anticipated session records should work.)

The Apache::Session interface

To use the `perl_session` table in a script, include the MySQL-related session module:

```
use Apache::Session::MySQL;
```

Apache::Session represents session information using a hash. It uses Perl's `tie` mechanism to map hash operations onto the storage and retrieval methods used by the underlying storage manager. Thus, to open a session, declare a hash variable and pass it to `tie`. The other arguments to `tie` are the name of the session module, the session ID, and a hashref containing information about the database to use. There are two ways to specify the database connection. One method passes a reference to a hash that contains connection parameters (and the session table name if you do not use the default name of `sessions`):

```
my %session;
tie %session,
    "Apache::Session::MySQL",
    $sess_id,
    {
      DataSource     => "DBI:mysql:host=localhost;database=cookbook",
      UserName       => "cbuser",
      Password       => "cbpass",
      LockDataSource => "DBI:mysql:host=localhost;database=cookbook",
      LockUserName   => "cbuser",
      LockPassword   => "cbpass",
      TableName      => "perl_session"
    };
```

In this case, Apache::Session uses the parameters to open its own connection to MySQL, which it closes when you close or destroy the session.

The other method passes the handle for an already open database connection (represented here by `$dbh`):

```
my %session;
tie %session,
    "Apache::Session::MySQL",
    $sess_id,
    {
      Handle     => $dbh,
      LockHandle => $dbh,
      TableName  => "perl_session"
    };
```

If you pass a handle to an open connection as just shown, Apache::Session leaves it open when you close or destroy the session, on the assumption that you're using the handle for other purposes elsewhere in the script. Close the connection yourself when you're done with it.

The `$sess_id` argument to `tie` is the session identifier. Its value is either `undef` to begin a new session, or the ID of an existing session record (a value that matches the `id` column in some existing `perl_session` table row).

After the session has been opened, you can access its contents. For example, you'll want to determine its identifier so that you can send it to the client:

```
$sess_id = $session{_session_id};
```

Apache::Session reserves for internal use hash element names that begin with an underscore, such as `_session_id`. Other than that, you can use choose your own names for storing session values.

To save a scalar value in the session, store it by value. To access a scalar, read the value directly. For example, maintain a scalar counter value as follows, where the counter is initialized if the session is new, and then incremented and retrieved for display:

```
$session{count} = 0 if !exists ($session{count}); # initialize counter
++$session{count};                                # increment counter
print "counter value: $session{count}\n";         # print value
```

To save a nonscalar value such as an array or a hash into the session record, store a reference to it:

```
$session{my_array} = \@my_array;
$session{my_hash} = \%my_hash;
```

In this case, changes made to `@my_array` or `%my_hash` before you close the session will be reflected in the session contents. To save an independent copy of an array or hash in the session that will not change when you modify the original, create a reference to the copy like this:

```
$session{my_array} = [ @my_array ];
$session{my_hash} = { %my_hash };
```

To retrieve a nonscalar value, dereference the reference stored in the session:

```
@my_array = @{$session{my_array}};
%my_hash = %{$session{my_hash}};
```

To close a session when you're done with it, pass it to `untie`:

```
untie (%session);
```

When you close a session, Apache::Session saves it to the `perl_session` table if you've made changes to it. This also makes the session values inaccessible, so don't close the session until you're done with it.

Apache::Session notices changes to "top-level" session record values, but might not detect a change to a member of a value stored by reference (such as an array element). If this is a problem, you can force Apache::Session to save a session when you close it by assigning any top-level session element a value. The session ID is always present in the session hash, so the following idiom provides a convenient way to force session saving:

```
$session{_session_id} = $session{_session_id};
```

An open session can be terminated rather than closed. Doing so removes the corresponding row from the `perl_session` table so that it can no longer be used:

```
tied (%session)->delete ();
```

A sample Apache::Session application

The following script, *sess_track.pl*, is a short but complete session application. It uses Apache::Session to track the number of requests in the session and the time of each request, updating and displaying the information each time it is invoked. *sess_track.pl* uses the CGI.pm cookie management interface to pass the session ID in a cookie named `PERLSESSID`:

```
#!/usr/bin/perl
# sess_track.pl: session request counting/timestamping demonstration

use strict;
use warnings;
use CGI qw(:standard);
use Cookbook;
use Apache::Session::MySQL;

my $title = "Perl Session Tracker";

my $dbh = Cookbook::connect ();       # connection to MySQL
my $sess_id = cookie ("PERLSESSID");  # session ID (undef if new session)
my %session;                          # session hash
my $cookie;                           # cookie to send to client

# open the session

tie %session,
    "Apache::Session::MySQL",
    $sess_id,
    {
      Handle     => $dbh,
      LockHandle => $dbh,
      TableName  => "perl_session"
    };
if (!defined ($sess_id))          # this is a new session
{
  # get new session ID, initialize session data, create cookie for client
  $sess_id = $session{_session_id};
  $session{count} = 0;            # initialize counter
  $session{timestamp} = [];       # initialize timestamp array
  $cookie = cookie (-name => "PERLSESSID", -value => $sess_id);
}

# increment counter and add current timestamp to timestamp array

++$session{count};
push (@{$session{timestamp}}, scalar (localtime (time ())));
```

```
# construct content of page body

my $page_body =
    p ("This session has been active for $session{count} requests.")
    . p ("The requests occurred at these times:")
    . ol (li ($session{timestamp}))
    . p ("Reload page to send next request.");

if ($session{count} < 10) # close (and save) session
{
  untie (%session);
}
else                      # destroy session after 10 invocations
{
  tied (%session)->delete ();
  # reset cookie to tell browser to discard session cookie
  $cookie = cookie (-name    => "PERLSESSID",
                    -value   => $sess_id,
                    -expires => "-1d");   # "expire yesterday"
}

$dbh->disconnect ();

# generate the output page; include cookie in headers if it's defined

print header (-cookie => $cookie)
      . start_html (-title => $title)
      . $page_body
      . end_html ();
```

For information about CGI.pm cookie support, use the following command and read the section describing the `cookie()` function:

```
% perldoc CGI
```

To try the script, install it in your web server's *cgi-bin* directory and request it from your browser. To reinvoke it, use your browser's Reload function.

sess_track.pl opens the session and increments the counter prior to generating any page output. This is necessary because the client must be sent a cookie containing the session name and identifier if the session is new. Any cookie sent must be part of the response headers, so the page body cannot be sent until after the headers.

The script saves the page content in a variable rather than writing it immediately. Should the session need to be terminated, the script resets the cookie to be one that tells the browser to discard the one it has. This must be determined prior to sending any page content.

Session expiration

The Apache::Session module requires only the `id` and `a_session` columns in the `perl_session` table. The module makes no provision for timing out or expiring sessions, but doesn't restrict you from adding other columns, so you can implement those capabilities yourself. Add a `TIMESTAMP` column to the table to store the time of each session's last modification (MySQL updates it automatically whenever a session record is changed):

```
ALTER TABLE perl_session
ADD update_time TIMESTAMP NOT NULL,
ADD INDEX (update_time);
```

To expire sessions, run a statement periodically that sweeps the table and removes old rows. The following statement uses an expiration time of four hours:

```
DELETE FROM perl_session WHERE update_time < NOW() - INTERVAL 4 HOUR;
```

The `ALTER TABLE` statement indexes `update_time` to make the `DELETE` operation faster.

To expire rows automatically, create a scheduled event (see Recipe 9.8). This event runs every four hours:

```
CREATE EVENT expire_perl_session
  ON SCHEDULE EVERY 4 HOUR
  DO DELETE FROM perl_session WHERE update_time < NOW() - INTERVAL 4 HOUR;
```

21.2. Using MySQL-Based Storage in Ruby Applications

Problem

You want to use session storage for Ruby scripts.

Solution

Use the `CGI::Session` class interface. By default, it uses temporary files for backing store, but you can configure it to use MySQL instead.

Discussion

The `CGI::Session` class manages session storage. It identifies sessions using cookies, which it adds transparently to the responses sent to the client. `CGI::Session` permits use of alternative storage-management classes in place of the default manager that uses temporary files. We'll use the `mysql-session` package, which is based on the Ruby DBI interface and stores session records using MySQL. `mysql-session` is available from the *MySQL Cookbook* companion website (see the Preface).

To use `mysql-session` in a script, include these modules:

```
require "cgi"
require "cgi/session"
require "mysqlstore"
```

To create a session, first create a `CGI` object. Then invoke `CGI::Session.new`, which takes several arguments. The first is a `CGI` object associated with the script; it must exist before you can open the session. Other arguments provide information about the session itself. Those following are relevant no matter which storage manager you use:

`session_key`
: The session key is the name of the cookie to send to the client. The default key value is `_session_key`; we'll use `RUBYSESSID`.

`new_session`
: This argument is true to force a new session to be created, or false to use an existing session, which is assumed to have already been created during a previous request. It's also possible to create a session if it does not exist and use the current session if it does. To enable that behavior, omit the `new_session` argument; our example script does so.

`database_manager`
: The name of the class that provides storage management for session records. If this argument is omitted, the session manager uses temporary files.

To use the `mysql-session` package as the storage manager, the `database_manager` argument should be `CGI::Session::MySQLStore`. In that case, `mysql-session` enables several other arguments for the `CGI::Session.new` method. You can pass in arguments that instruct the session manager to establish its own connection to MySQL, or open your own connection and pass its database handle to the session manager.

The following discussion shows both approaches, but either way, we need a table for storing session records. For `mysql-session`, create a table named `ruby_session` with the following structure:

```
CREATE TABLE ruby_session
(
  session_id    VARCHAR(255) NOT NULL,
  session_value LONGBLOB NOT NULL,
  update_time   DATETIME NOT NULL,
  PRIMARY KEY (session_id)
);
```

To have the session manager open its own connection to MySQL, create the session like this:

```
cgi = CGI.new("html4")
sess_id = cgi.cookies["RUBYSESSID"]
session = CGI::Session.new(
            cgi,
            "session_key"      => "RUBYSESSID",
```

```
    "database_manager" => CGI::Session::MySQLStore,
    "db.host"          => "localhost",
    "db.user"          => "cbuser",
    "db.pass"          => "cbpass",
    "db.name"          => "cookbook",
    "db.table"         => "ruby_session",
    "db.hold_conn"     => 1
  )
```

The `db.xxx` parameters used in that code tell `mysql-session` how to connect to the server, as well as the database and table for session records:

`db.host`
: The host where the MySQL server is running.

`db.user`, `db.pass`
: The username and password of the MySQL account to use.

`db.name`, `db.table`
: The database and table names for the session table.

`db.hold_conn`
: By default, `mysql-session` opens and closes a connection each time it needs to send a statement to the MySQL server. If the `db.hold_conn` parameter is 1, `mysql-session` opens the connection only once and holds it open until the session ends.

Another way to create a session is to pass the handle for an already open database connection(represented using `dbh`):

```
cgi = CGI.new("html4")
sess_id = cgi.cookies["RUBYSESSID"]
session = CGI::Session.new(
            cgi,
            "session_key"      => "RUBYSESSID",
            "database_manager" => CGI::Session::MySQLStore,
            "db.dbh"           => dbh,
            "db.name"          => "cookbook",
            "db.table"         => "ruby_session"
          )
```

In this case, the `db.host`, `db.user`, `db.pass`, and `db.hold_conn` parameters are not used. Close the connection yourself when you're done with it.

Whichever way you create the session, its ID is available while it is open as the `session.session_id` attribute.

To close or destroy the session, invoke the `close` or `delete` method of the session object, respectively.

The session manager stores data as key/value pairs, using strings for the values. It does not know the types of the values that you store. I find the following strategy useful for dealing with type-conversion issues:

1. After opening the session, extract values from the session and convert them from "generic" string form to properly typed values.
2. Work with the typed values until it is time to close the session.
3. Convert the typed values to string form, store them in the session, and close it.

The following script uses the `CGI::Session` session manager to track the number of requests in a session and the time of each request. After 10 requests, the script deletes the session to cause a new session to begin for the next request:

```
#!/usr/bin/ruby -w
# sess_track.rb: session request counting/timestamping demonstration

require "Cookbook"
require "cgi"
require "cgi/session"
require "mysqlstore"

title = "Ruby Session Tracker";

dbh = Cookbook::connect

cgi = CGI.new("html4")
session = CGI::Session.new(
            cgi,
            "session_key"      => "RUBYSESSID",
            "database_manager" => CGI::Session::MySQLStore,
            "db.dbh"           => dbh,
            "db.name"          => "cookbook",
            "db.table"         => "ruby_session"
          )

# extract string values from session, convert them to the proper types

count = session["count"]
count = (count.nil? ? 0 : count.to_i)
timestamp = session["timestamp"]
timestamp = (timestamp.nil? ? [] : timestamp.split(","))

# increment counter and add current timestamp to timestamp array

count = count + 1
timestamp << Time.now().strftime("%Y-%m-%d %H:%M:%S")

# construct content of page body

page = ""

page << cgi.p {"This session has been active for #{count} requests."}
page << cgi.p {"The requests occurred at these times:"}
page << cgi.ol { timestamp.collect { |t| cgi.li { t.to_s } } }
```

```
page << cgi.p {"Reload page to send next request."}

if count < 10     # save modified values into session
  # convert session variables back to strings before saving
  session["count"] = count.to_s
  session["timestamp"] = timestamp.join(",")
  session.close()
else              # destroy session after 10 invocations
  session.delete()
end

dbh.disconnect

# generate the output page

cgi.out {
  cgi.html {
    cgi.head { cgi.title { title } } + cgi.body() { page }
  }
}
```

`CGI::Session` makes no provision for expiring sessions, but you can discard old session records using a technique similar to that discussed in Recipe 21.1. Should you do this, index the `update_time` column to make `DELETE` statements faster:

```
ALTER TABLE ruby_session ADD INDEX (update_time);
```

21.3. Using MySQL-Based Storage with the PHP Session Manager

Problem

You want to use session storage for PHP scripts.

Solution

PHP includes session management. By default, it uses temporary files for backing store, but you can configure it to use MySQL instead.

Discussion

This section shows how to use the PHP native session manager and how to extend it by implementing a storage module that saves session data in MySQL. If your PHP configuration has the `track_vars` configuration variable enabled (which it is by default), session variables are available as elements of the `$_SESSION` superglobal array. If the `register_globals` configuration variable is enabled as well, session variables also exist in your script as global variables of the same names. This is less secure, so this variable is

assumed *not* to be enabled here. (Recipe 20.5 discusses the security implications of `register_globals`, although it is no longer an issue as of PHP 5.4, when it was removed.)

The PHP session management interface

PHP's session management is based on a small number of functions, all of which are documented in the PHP manual. The following list describes those we use:

`session_start ()`
: Opens a session and extracts any variables previously stored in it, making them available in the script's global namespace. For example, a session variable named `x` becomes available as `$_SESSION["x"]`. This function *must* be called first before using the relevant session variable array.

`session_write_close ()`
: Writes the session data and closes the session. The PHP documentation indicates that normally you need not call this function because PHP saves an open session automatically when your script ends. However, it appears that in PHP 5, that might not always be true when you provide your own session handler. To be safe, call this function to save your changes.

`session_destroy ()`
: Removes the session and any data associated with it.

Specifying a user-defined storage module

The PHP session management interface just described specifies nothing about backing store or how session information actually gets saved. By default, PHP uses temporary files to store session data, but the session interface is extensible to permit other storage modules. To override the default storage method and store session data in MySQL, do this:

1. Set up a table to hold session records, and write the routines that implement the storage module. These actions are done once, prior to writing any scripts that use the new module.
2. Tell PHP that you're supplying a user-defined storage manager. Do this globally in *php.ini* (in which case you make the change once), or within individual scripts (in which case it's necessary to declare your intent in each script).
3. Register the storage module routines within each script that uses the module.

Creating the session table. Any MySQL-based storage module needs a table in which to store session information. Create a table named `php_session` defined as follows:

```
CREATE TABLE php_session
(
  id          CHAR(32) NOT NULL,
  data        LONGBLOB,
  update_time TIMESTAMP NOT NULL,
  PRIMARY KEY (id),
  INDEX (update_time)
);
```

The `id` column holds session identifiers, which are unique 32-character MD5 values. The `data` column holds session information. PHP serializes session data into a string before storing it, so `php_session` needs only a large generic string column to hold the resulting serialized value. The `update_time` column is a `TIMESTAMP`, so MySQL updates it automatically whenever a session record is updated. This column is not required by PHP, but it's useful for implementing a garbage collection policy based on each session's most recent update time.

A small number of statements suffice to manage the contents of the `php_session` table as we have defined it:

- To retrieve a session's data, use a simple `SELECT` based on the session identifier:

  ```
  SELECT data FROM php_session WHERE id = 'sess_id';
  ```

- To write session data, a `REPLACE` updates an existing row, or creates a new one if no such row exists:

  ```
  REPLACE INTO php_session (id,data) VALUES('sess_id','sess_data');
  ```

  `REPLACE` also updates the timestamp in the row when creating or updating a row, which is important for garbage collection.

  Some storage manager implementations use a combination of `INSERT` and a fallback to `UPDATE` if the `INSERT` fails because a row with the given session ID already exists (or an `UPDATE` with a fallback to `INSERT` if the `UPDATE` fails because a row with the ID does *not* exist). In MySQL, `REPLACE` performs the required task with a single statement.

- To destroy a session, delete the corresponding row:

  ```
  DELETE FROM php_session WHERE id = 'sess_id';
  ```

- To perform garbage collection, remove old rows. The following statement deletes rows that have a timestamp value more than *`sess_life`* seconds old:

  ```
  DELETE FROM php_session
  WHERE update_time < NOW() - INTERVAL sess_life SECOND;
  ```

  The PHP session manager supplies the value of *`sess_life`* when it invokes the garbage collection routine. (The table definition for `php_session` indexes `update_time` to make `DELETE` statements faster.)

These statements form the basis of the routines that make up our MySQL-backed storage module. The primary function of the module is to open and close MySQL connections and issue the proper statements at the appropriate times.

Writing the storage management routines. A user-defined session storage module is implemented as a set of handler routines. To register them with PHP's session manager, call `session_set_save_handler()`, where each argument is a handler routine name specified as a string:

```
session_set_save_handler (
  "mysql_sess_open",      # function to open a session
  "mysql_sess_close",     # function to close a session
  "mysql_sess_read",      # function to read session data
  "mysql_sess_write",     # function to write session data
  "mysql_sess_destroy",   # function to destroy a session
  "mysql_sess_gc"         # function to garbage-collect old sessions
);
```

The order of the handler routines must be as shown, but you can name them as you like. They need not necessarily be named `mysql_sess_open()`, `mysql_sess_close()`, and so forth. Write the routines according to the following specifications:

`mysql_sess_open ($save_path, $sess_name)`
: Performs any actions necessary to begin a session. `$save_path` is the name of the location where sessions should be stored; this is useful for file storage only. `$sess_name` indicates the name of the session identifier (for example, `PHPSESSID`). A MySQL-based storage manager can ignore both arguments. The function returns `TRUE` or `FALSE` to indicate whether the session was opened successfully.

`mysql_sess_close ()`
: Closes the session, returning `TRUE` for success or `FALSE` for failure.

`mysql_sess_read ($sess_id)`
: Retrieves the data associated with the session identifier and returns it as a string. If there is no such session, the function returns an empty string. If an error occurs, it returns `FALSE`.

`mysql_sess_write ($sess_id, $sess_data)`
: Saves the data associated with the session identifier, returning `TRUE` for success or `FALSE` for failure. PHP itself takes care of serializing and unserializing the session contents, so the read and write functions need deal only with serialized strings.

`mysql_sess_destroy ($sess_id)`
: Destroys the session and any data associated with it, returning `TRUE` for success or `FALSE` for failure. For MySQL-based storage, destroying a session amounts to deleting the row from the `php_session` table associated with the session ID.

`mysql_sess_gc ($gc_maxlife)`
: Performs garbage collection to remove old sessions. This function is invoked on a probabilistic basis. When PHP receives a request for a page that uses sessions, it calls the garbage collector with a probability defined by the `session.gc_probability` configuration variable in *php.ini*. For example, if the probability value is 1 (that is, 1%), PHP calls the collector approximately once every hundred requests. If the value is 100, it calls the collector for every request—probably more processing overhead than you'd want.

  The argument to `gc()` is the maximum session lifetime in seconds. Sessions older than that are considered subject to removal. The function returns `TRUE` for success or `FALSE` for failure.

To register the handler routines, call `session_set_save_handler()`, which should be done in conjunction with informing PHP that you'll be using a user-defined storage module. The default storage management method is defined by the `session.save_handler` configuration variable. You can change the method globally by modifying the *php.ini* initialization file, or within individual scripts:

- To change the storage method globally, edit *php.ini*. The default configuration setting specifies the use of file-based session storage management:

  ```
  session.save_handler = files;
  ```

  Modify this to indicate that sessions will be handled by a user-level mechanism:

  ```
  session.save_handler = user;
  ```

  If you use PHP as an Apache module, restart Apache after modifying *php.ini* so that PHP notices the changes.

  If you change the storage method globally, every PHP script that uses sessions will be expected to provide its own storage management routines. This may have unintended side effects for other script writers if they are unaware of the change. For example, other developers that use the web server may want to continue using file-based sessions.
- As an alternative to making a global change, specify a different storage method by calling `ini_set()` on a per-script basis:

  ```
  ini_set ("session.save_handler", "user");
  ```

  `ini_set()` is less intrusive than a global configuration change. The storage manager we develop here uses `ini_set()`, to trigger database-backed session storage only for those scripts that request it.

To make it easy to access an alternative session storage module, it's useful to create a library file, *Cookbook_Session.php*. The only thing a script need do to use the library file is include it prior to starting the session. The outline of the file looks like this:

```
<?php
# Cookbook_Session.php: MySQL-based session storage module

require_once "Cookbook.php";

# Define the handler routines

function mysql_sess_open ($save_path, $sess_name) ...
function mysql_sess_close () ...
function mysql_sess_read ($sess_id) ...
function mysql_sess_write ($sess_id, $sess_data) ...
function mysql_sess_destroy ($sess_id) ...
function mysql_sess_gc ($gc_maxlife) ...

# Initialize the connection identifier, select user-defined
# session handling, and register the handler routines

$mysql_sess_dbh = NULL;
ini_set ("session.save_handler", "user");
session_set_save_handler (
  "mysql_sess_open",
  "mysql_sess_close",
  "mysql_sess_read",
  "mysql_sess_write",
  "mysql_sess_destroy",
  "mysql_sess_gc"
);
?>
```

The library file includes *Cookbook.php* so that it can access the connection routine for opening a connection to the cookbook database. Then it defines the handler routines (we'll get to the details of these functions shortly). Finally, it initializes the connection identifier, tells PHP to get ready to use a user-defined session storage manager, and registers the handler functions. Thus, a PHP script that wants to store sessions in MySQL performs all the necessary setup simply by including the *Cookbook_Session.php* file:

```
require_once "Cookbook_Session.php";
```

The interface provided by the *Cookbook_Session.php* library file exposes a global database connection identifier variable (`$mysql_sess_conn`) and a set of handler routines named `mysql_sess_open()`, `mysql_sess_close()`, and so forth. Scripts that use the library should avoid using these global names for other purposes.

Now let's implement each handler routine.

Opening a session. PHP passes two arguments to this function: the save path and the session name. The save path applies to file-based storage, and we don't need the session

name, so both arguments can be ignored. The function therefore does nothing but open a connection to MySQL:

```
function mysql_sess_open ($save_path, $sess_name)
{
global $mysql_sess_dbh;

  # open connection to MySQL if it's not already open
  if ($mysql_sess_dbh === NULL)
  {
    try
    {
      $mysql_sess_dbh = Cookbook::connect ();
    }
    catch (PDOException $e)
    {
      $mysql_sess_dbh = NULL;
      return (FALSE);
    }
  }
  return (TRUE);
}
```

Closing a session. The close handler checks whether a connection to MySQL is open and closes it if so:

```
function mysql_sess_close ()
{
global $mysql_sess_dbh;

  if ($mysql_sess_dbh !== NULL)    # close connection if it's open
    $mysql_sess_dbh = NULL;
  return (TRUE);
}
```

Reading session data. The `mysql_sess_read()` function uses the session ID to look up the data for the corresponding session record and returns it. It returns the empty string if no such record exists, or `FALSE` if an error occurs:

```
function mysql_sess_read ($sess_id)
{
global $mysql_sess_dbh;

  try
  {
    $stmt = "SELECT data FROM php_session WHERE id = ?";
    $sth = $mysql_sess_dbh->prepare ($stmt);
    $sth->execute (array ($sess_id));
    list ($data) = $sth->fetch (PDO::FETCH_NUM);
    if (isset ($data))
      return ($data);
```

```
  }
  catch (PDOException $e) { /* do nothing */ }
  return ("");
}
```

Writing session data. `mysql_sess_write()` creates a new record if there is none for the session yet, or replaces the existing record if there is one:

```
function mysql_sess_write ($sess_id, $sess_data)
{
global $mysql_sess_dbh;

  try
  {
    $stmt = "REPLACE php_session (id, data) VALUES(?,?)";
    $sth = $mysql_sess_dbh->prepare ($stmt);
    $sth->execute (array ($sess_id, $sess_data));
    return (TRUE);
  }
  catch (PDOException $e)
  {
    return (FALSE);
  }
}
```

Destroying a session. When a session is no longer needed, `mysql_sess_destroy()` removes the corresponding record:

```
function mysql_sess_destroy ($sess_id)
{
global $mysql_sess_dbh;

  try
  {
    $stmt = "DELETE FROM php_session WHERE id = ?";
    $sth = $mysql_sess_dbh->prepare ($stmt);
    $sth->execute (array ($sess_id));
    return (TRUE);
  }
  catch (PDOException $e)
  {
    return (FALSE);
  }
}
```

Performing garbage collection. The `TIMESTAMP` column `update_time` in each session record indicates when the session was last updated. `mysql_sess_gc()` uses this value to implement garbage collection. The argument `$sess_maxlife` specifies how old sessions can be (in seconds). Older sessions are considered expired and candidates for removal,

which is easily done by deleting session records having a timestamp older than the current time more than the permitted lifetime:

```
function mysql_sess_gc ($sess_maxlife)
{
global $mysql_sess_dbh;

  try
  {
    $stmt = "DELETE FROM php_session
             WHERE update_time < NOW() - INTERVAL ? SECOND";
    $sth = $mysql_sess_dbh->prepare ($stmt);
    $sth->execute (array ($sess_maxlife));
  }
  catch (PDOException $e) { /* do nothing */ }
  return (TRUE);  # ignore errors
}
```

Using the storage module. Install the *Cookbook_Session.php* file in a public library directory accessible to your scripts. On my system, I put PHP library files in */usr/local/lib/mcb* and modify *php.ini* so that the `include_path` variable names that directory (see Recipe 2.3). To try the storage module, install the following example script, *sess_track.php*, in your web tree and invoke it a few times to see how the information display changes:

```
<?php
# sess_track.php: session request counting/timestamping demonstration

require_once "Cookbook_Session.php";
require_once "Cookbook_Webutils.php"; # for make_ordered_list()

$title = "PHP Session Tracker";

# Open session and extract session values

session_start ();
$count = $_SESSION["count"];
$timestamp = $_SESSION["timestamp"];

# If the session is new, initialize the variables

if (!isset ($count))
  $count = 0;
if (!isset ($timestamp))
  $timestamp = array ();

# Increment counter, add current timestamp to timestamp array

++$count;
$timestamp[] = date ("Y-m-d H:i:s T");
```

```
if ($count < 10)  # save modified values into session
{
  $_SESSION["count"] = $count;
  $_SESSION["timestamp"] = $timestamp;
  session_write_close (); # save session changes
}
else              # destroy session after 10 invocations
{
  session_destroy ();
}

# Produce the output page
?>
<html>
<head><title><?php print ($title); ?></title></head>
<body>

<?php
print ("<p>This session has been active for $count requests.</p>");
print ("<p>The requests occurred at these times:</p>");
print make_ordered_list ($timestamp);
print ("<p>Reload page to send next request.</p>");
?>

</body>
</html>
```

The script includes the *Cookbook_Session.php* library file to enable the MySQL-based storage module, then uses the PHP session manager interface in typical fashion. First, it opens the session and attempts to extract the session variables. For the first request, the session variables are not set and must be initialized. This is determined by the `isset()` tests. The scalar variable `$count` starts out at zero, and the nonscalar variable `$timestamp` starts out as an empty array. For successive requests, the session variables have the values assigned to them by the previous request.

Next, the script increments the counter, adds the current timestamp to the end of the timestamp array, and calls `session_write_close()` to write the changes to session data. If the session limit of 10 invocations has been reached, the script destroys the session. This causes the session to restart on the next request.

After updating the session data, *sess_track.php* produces an output page that displays the count and the access times.

The output page is produced after updating the session record because PHP might determine that a cookie containing the session ID must be sent to the client. That determination must be made before generating the page body because cookies are sent in the headers.

To access session variables, use the `$_SESSION` superglobal array after calling `session_start()`. For example, the session variable named `count` is available as `$_SESSION["count"]`.

21.4. Using MySQL for Session-Backing Store with Tomcat

Problem

You want to use session storage for Java-based scripts.

Solution

Tomcat handles session management for you. By default, it uses temporary files for backing store. To configure it to use MySQL instead, modify the appropriate Tomcat configuration file to supply JDBC parameters.

Discussion

The Perl, Ruby, and PHP session mechanisms described earlier in this chapter require applications to indicate explicitly that they want to use MySQL-based session storage. For Perl and Ruby, a script must state that it wants to use the appropriate session module. For PHP, the session manager is built into the language, but each application that uses a MySQL storage module must register it.

For Java applications that run under Tomcat, a different framework applies. Tomcat itself manages sessions, so to store session information in MySQL, reconfigure Tomcat, not your applications. This relieves web-based Java programs of some of the messy session-related details handled at the application level in other languages. For example, the Tomcat server rather than your application handles session IDs. If cookies are enabled, Tomcat uses them. Otherwise, it uses URL rewriting to encode the session ID in the URL. Application developers need not care which method Tomcat uses because the ID is available the same way for either method.

To illustrate the independence of applications from the session management method used by Tomcat, this section shows simple JSP application scripts that use a session. Then it shows how to reconfigure Tomcat to store session information in MySQL rather than in the default session store—without requiring any changes at all to the application scripts.

This section assumes that the `mcb` application has been installed into and unpacked under the Tomcat *webapps* directory (see Recipe 18.3). For background on Tomcat itself, read "JSP, JSTL, and Tomcat Primer" on the companion website (see the Preface).

The servlet and JSP session interface

Tomcat uses the standard session interface described in the Java Servlet Specification. This interface can be used both by servlets and by JSP pages. Within a servlet, access the session by importing the `javax.servlet.http.HttpSession` class and invoking the `getSession()` method of your `HttpRequest` object:

```
import javax.servlet.http.*;
HttpSession session = request.getSession ();
```

In JSP pages, session support is enabled by default, so it's as though those statements have already been issued by the time the page begins executing. The session is available implicitly through a `session` variable that's already set up for you.

The `HttpSession` section of the Java Servlet Specification defines the complete session interface. Some representative session object methods are listed here:

`isNew ()`
: Returns true or false to indicate whether the session began with the current request.

`getAttribute (String attrName)`
: Session contents consist of attributes, which are objects bound to names. To access a session attribute, specify its name. `getAttribute()` returns the `Object` bound to the named session attribute, or `null` if there is no object with that name.

`setAttribute (String attrName, Object obj)`
: Adds the object to the session and binds it to the given name.

`removeAttribute (String attrName)`
: Removes the named attribute from the session.

`invalidate ()`
: Invalidates the session and any data associated with it. The next request from the client will begin a new session.

A sample JSP session application

The following example shows a JSP page, *sess_track.jsp*, that maintains a session request counter and a log of the request times. To illustrate the session-related operations more explicitly, this page consists primarily of embedded Java code that uses the `HttpSession` session interface directly:

```
<%--
  sess_track.jsp: session request counting/timestamping demonstration
--%>

<%@ page import="java.util.*" %>
<%
  // get session variables, initializing them if not present

  int count;
```

```
  Object obj = session.getAttribute ("count");
  if (obj == null)
    count = 0;
  else
    count = Integer.parseInt (obj.toString ());

  ArrayList timestamp = (ArrayList) session.getAttribute ("timestamp");
  if (timestamp == null)
    timestamp = new ArrayList ();

  // increment counter, add current timestamp to timestamp array

  count = count + 1;
  timestamp.add (new Date ());

  if (count < 10)   // save updated values in session object
  {
    session.setAttribute ("count", String.valueOf (count));
    session.setAttribute ("timestamp", timestamp);
  }
  else              // restart session after 10 requests
  {
    session.invalidate ();
  }
%>

<html>
<head><title>JSP Session Tracker</title></head>
<body>

<p>This session has been active for <%= count %> requests.</p>
<p>The requests occurred at these times:</p>
<ol>
<%
  for (int i = 0; i < timestamp.size (); i++)
    out.println ("<li>" + timestamp.get (i) + "</li>");
%>
</ol>
<p>Reload page to send next request.</p>

</body>
</html>
```

sess_track.jsp is included in the `mcb` application (see Recipe 18.3). Invoke it from your browser and reload it a few times to see how the display changes.

The `session.setAttribute()` method used in *sess_track.jsp* places information into the session so that it can be found by later invocations of the script. But session attributes also can be shared with other scripts in the same application context, which have access to the same information. You'll see this with our next version of the script, which when you invoke it accesses the same session information as *sess_track.jsp*.

Some session-related operations shown in *sess_track.jsp* can be done using tags from the JSTL tag library, which provides a `sessionScope` variable for accessing the implicit JSP session object. The following script, *sess_track2.jsp*, uses that variable. One difference in approach is that *sess_track.jsp* terminates the session by calling `session.invalidate()`, but the `sessionScope` variable provides no access to that method. Instead, *sess_track2.jsp* terminates the session by deleting the session contents, causing the session to restart with the next client request:

```
<%--
  sess_track2.jsp: session request counting/timestamping demonstration
--%>

<%@ page import="java.util.*" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test="${empty sessionScope.count}">
  <c:set var="count" scope="session" value="0"/>
</c:if>
<c:set var="count" scope="session" value="${sessionScope.count+1}"/>

<%
  ArrayList timestamp = (ArrayList) session.getAttribute ("timestamp");
  if (timestamp == null)
    timestamp = new ArrayList ();
  // add current timestamp to timestamp array, store result in session
  timestamp.add (new Date ());
  session.setAttribute ("timestamp", timestamp);
%>

<html>
<head><title>JSP Session Tracker</title></head>
<body>

<p>This session has been active for
<c:out value="${sessionScope.count}"/>
requests.</p>
<p>The requests occurred at these times:</p>
<ol>
<c:forEach items="${sessionScope.timestamp}" var="t">
  <li><c:out value="${t}"/></li>
</c:forEach>
</ol>
<p>Reload page to send next request.</p>

<%-- has session limit of 10 requests been reached? --%>

<c:if test="${sessionScope.count ge 10}">
  <c:remove var="count" scope="session"/>
  <c:remove var="timestamp" scope="session"/>
</c:if>
```

```
</body>
</html>
```

Telling Tomcat to save session records in MySQL

The default Tomcat default session storage mechanism uses temporary files. To save sessions using JDBC with MySQL instead, follow this procedure:

1. Create a table to hold session records.
2. Make sure that Tomcat can access the proper JDBC driver.
3. Modify the appropriate Tomcat configuration file to specify use of a persistent session manager for the relevant application context.

None of these steps involve modifying the sample session script in any way, which reflects how Tomcat implements session support above the application level.

1. Create the Tomcat session table.

   Tomcat stores several types of information in the session table:

   - The session ID. By default, IDs are 32-character MD5 values.
   - The application name.
   - The session data. This is a serialized string.
   - Whether the session is valid, as a single byte.
   - The maximum permitted inactivity time, as a 32-bit integer measured in seconds.
   - The last access time, as a 64-bit integer.

   The following table satisfies those specifications; create it now before proceeding:

   ```
   CREATE TABLE tomcat_session
   (
     id            VARCHAR(32) NOT NULL,
     app           VARCHAR(255),
     data          LONGBLOB,
     valid_session CHAR(1) NOT NULL,
     max_inactive  INT NOT NULL,
     update_time   BIGINT NOT NULL,
     PRIMARY KEY (id),
     INDEX (app)
   );
   ```

2. Place the JDBC driver where Tomcat can find it.

   Because Tomcat itself manages sessions, it must be able to access the JDBC driver used to store sessions in a database. It's common to install drivers in the *lib* directory of the Tomcat tree so that they're available both to Tomcat and to applications. Install

the MySQL Connector/J driver there now if you haven't already (see Recipe 18.3). After a restart, Tomcat will be able to use it.

3. Modify the Tomcat configuration file.

   To tell Tomcat to use the `tomcat_session` table, modify the `mcb` application context file. Change location into the *webapps/mcb/META-INF* under the Tomcat *webapps* directory, copy *context.xml.jdbc* to *context.xml*, and restart Tomcat.

   If you look in *context.xml*, you'll find a `<Context>` element containing a `<Manager>` element that specifies the use of JDBC for MySQL-based session storage:

```
<Context path="/mcb" docBase="mcb" debug="0" reloadable="true">
  <Manager
    className="org.apache.catalina.session.PersistentManager"
    debug="0"
    saveOnRestart="true"
    maxIdleBackup="600">
    maxIdleSwap="1200"
    minIdleSwap="900"
    <Store
      className="org.apache.catalina.session.JDBCStore"
      driverName="com.mysql.jdbc.Driver"
      connectionURL=
        "jdbc:mysql://localhost/cookbook?user=cbuser&amp;password=cbpass"
      sessionTable="tomcat_session"
      sessionIdCol="id"
      sessionAppCol="app"
      sessionDataCol="data"
      sessionValidCol="valid_session"
      sessionMaxInactiveCol="max_inactive"
      sessionLastAccessedCol="update_time"
    />
  </Manager>
</Context>
```

The `<Manager>` element attributes specify general session-related options. Within the `<Manager>` element body, the `<Store>` element provides attributes pertaining to the JDBC driver. The following discussion focuses on the attributes shown in the example, but there are others you can use. For more information, see the Tomcat session-management documentation.

The `<Manager>` attributes shown in the example have the following meanings:

`className`
: The Java class that implements persistent session storage. It must be `org.apache.catalina.session.PersistentManager`.

`debug`
: The logging detail level. A value of zero disables debug output; higher numbers generate more output.

`saveOnRestart`
: Whether application sessions survive server restarts. Set it to `true` to have Tomcat save current sessions when it shuts down (and reload them when it starts up).

`maxIdleBackup`
: The number of seconds before inactive sessions are eligible for being saved to MySQL. A value of `-1` (the default) means "never."

`maxIdleSwap`
: The number of seconds before idle sessions should be swapped (saved to MySQL and passivated out of server memory). A value of `-1` (the default) means "never." If not `-1`, the value should be at least as great as `maxIdleBackup`.

`minIdleSwap`
: The number of seconds before idle sessions are eligible to be swapped. A value of `-1` (the default) means "never." If not `-1`, the value should be less than `maxIdleSwap`.

Within the `<Manager>` element, the `<Store>` element indicates how to connect to the database server, the names of the database and table for storing session records, and the names of the columns in the table:

`className`
: The name of a class that implements the `org.apache.catalina.Store` interface. For JDBC-based storage managers, the value is `org.apache.catalina.session.JDBCStore`.

`driverName`
: The class name for the JDBC driver. For the Connector/J driver, the value is `com.mysql.jdbc.Driver`.

`connectionURL`
: The URL for connecting to the database server, with characters that are special in XML properly encoded. The following URL connects to the MySQL server on the local host, using a database, username, and password of `cookbook`, `cbuser`, and `cbpass`, respectively. Notice that the `&` character that separates the `user` and `password` connection parameters is written as the `&amp;` entity:

```
jdbc:mysql://localhost/cookbook?user=cbuser&amp;password=cbpass
```

`sessionTable`
: The table in which to store session records. For our example, this is the `tomcat_session` table described earlier. (The database that contains the table appears in the `connectionURL` value.)

The remaining `<Store>` attributes in the example indicate the column names in the session table. These attributes are `sessionIdCol`, `sessionAppCol`, `sessionDataCol`, `sessionValidCol`, `sessionMaxInactiveCol`, and `sessionLastAccessedCol`, which correspond in the obvious way to columns of the `tomcat_session` table.

After you create *context.xml* and restart Tomcat, invoke the *sess_track.jsp* or *sess_track2.jsp* scripts a few times to initiate a session. Each should behave the same as before you reconfigured Tomcat, but now session information will be stored in MySQL. After a period of inactivity equal to the `<Manager>` element `maxIdleBackup` attribute value, you should see a session record appear in the `tomcat_session` table. If you watch the MySQL query log, you should also see sessions being saved to MySQL when you stop Tomcat.

Session expiration in Tomcat

Sessions persist for 30 minutes by default. To provide an explicit duration for a session manager, add a `maxInactiveInterval` attribute (in seconds) to your `<Manager>` element. To provide a duration specific to a particular application context, add a `<session-config>` element to the application's *WEB-INF/web.xml* file, specifying a timeout in minutes. For example, to use a value of 60 minutes:

```
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
```

If you modify either the file containing the `<Manager>` element or the *web.xml* file, restart Tomcat.

Session tracking in Tomcat

Although your JSP pages need do nothing to have Tomcat set up sessions or to use JDBC for session storage, they may need to take a small step to make sure that sessions move from request to request properly. This is necessary if you generate pages that contain hyperlinks to other pages that participate in the same session.

Tomcat generates a session identifier and tracks the session using cookies if it receives a cookie from the client that contains the session ID. If the client has cookies disabled, Tomcat tracks the session by rewriting URLs to include the session ID. You need not determine which method Tomcat is using, but you must ensure proper propagation of the session ID in case it is being passed by URL rewriting. When you create a page that includes a link to another page that is part of the session, do not list the path to the page like this:

```
To go to the next page,
<a href="nextpage.jsp">click here</a>.
```

That link doesn't contain the session ID. If Tomcat is tracking the session using URL rewriting, the ID is lost when the user selects the link. Instead, pass the link to `encodeURL()` to enable Tomcat to add the session ID to the URL as necessary:

```
To go to the next page,
<a href="<%= response.encodeURL ("nextpage.jsp") %>">click here</a>.
```

If Tomcat is tracking the session with cookies, `encodeURL()` returns the URL unchanged. However, if Tomcat is tracking the session by means of URL rewriting, `encodeURL()` adds the session ID to the page path automatically, so that it looks something like this:

```
mypage.jsp;jsessionid=xxxxxxxxxxxxxxx
```

Generate URLs using `encodeURL()` like this for links in any tag that takes the user to a page in the current session. This includes `<a>`, `<form>`, and `<frame>` tags, and possibly `<img>` tags, if those tags invoke a script that generates images on a session-specific basis.

It's probably best to develop the habit of using `encodeURL()` as a matter of routine when writing URLs for session-based applications. Even if you think everyone who uses the application will have cookies enabled, your assumption may prove incorrect some day.

The `java.net.URLEncoder.encode()` method has a name similar to `encodeURL()`, but it's different. It performs conversion of special characters to `%xx` notation to make them safe for use in URLs.

CHAPTER 22

Server Administration

22.0. Introduction

This chapter covers how to perform operations involved in administering a MySQL server:

- General server configuration
- The plug-in interface
- Controlling server logging
- Server monitoring
- Backup and recovery

The chapter doesn't cover managing MySQL user accounts. That is an administrative task, but is covered in Chapter 23.

Many of the techniques shown here require administrative access, such as the ability to modify tables in the `mysql` system database or use statements that require the `SUPER` privilege. For this reason, to carry out the operations described here, you'll likely need to connect to the server as `root` rather than as `cbuser`.

22.1. Configuring the Server

Problem

You want to change the server settings, and also verify that your changes took effect.

Solution

To change settings, specify them at server startup or at runtime. To verify the changes, examine the relevant system variables at runtime.

Discussion

The MySQL server places many configuration parameters under your control. For example, resources that require memory can be adjusted up or down to tailor resource usage. A heavily used server requires more memory; a lightly used one, less. You can set command options and system variables at server startup, and many system variables are settable at runtime as well. You can also examine your settings at runtime to verify that the configuration is as you intend.

Configuration control at server startup

To configure the server at startup time, specify options on the command line or in an option file. The latter is usually preferable because you can specify settings once and they'll apply at each startup. (For background on using command-line options and option files, see Recipe 1.4.)

Command option names typically use dashes, whereas system variable names use underscores. However, the server is more permissive at startup and recognizes command options and system variables written using dashes or underscores interchangeably. For example, `sql_mode` and `sql-mode` are equivalent on the command line or in an option file. This differs from runtime, when references to system variables *must* be written using underscores.

To specify server parameters in an option file, list them in the `[mysqld]` group of a file the server reads. To illustrate, here are some parameters you might set:

- The default character set is `latin1`. To use a more general character set, change this to `utf8`, or even `utf8mb4` to include the 4-byte supplemental characters not part of `utf8`.
- The default SQL mode is `NO_ENGINE_SUBSTITUTION` (empty before MySQL 5.6.6). To be more restrictive by default, enable strict SQL mode. Or be even more restrictive, like "traditional" database servers.
- The event scheduler is disabled by default. If you plan to use scheduled events (see Recipe 9.8), you must enable it.
- If your users run a lot of queries on InnoDB tables, it might be a good idea to increase the InnoDB buffer pool size from its default of 128MB.

To implement these configuration ideas, write the `[mysqld]` group in your option file like this:

```
[mysqld]
character_set_server=utf8
sql_mode=TRADITIONAL
event_scheduler=1
innodb_buffer_pool_size=256M
```

Those are just suggestions; adjust the server configuration for your own requirements. For information about plug-in and logging options in particular, see Recipes 22.2 and 22.3.

Configuration control and verification at runtime

After the server starts, you can make runtime adjustments by changing system variables using the `SET` statement:

```
SET GLOBAL var_name = value;
```

That statement sets the global value of *var_name*; that is, the value that applies to all clients by default. Changes to the global value at runtime require the `SUPER` privilege. Many system variables also have a session value, which is the value specific to a particular client session. The session value of a given variable is initialized from the global value when the client connects, but the client can change it thereafter. For example, the DBA might set the sort buffer size at server startup:

```
[mysqld]
sort_buffer_size=512K
```

That sets the global value. A DBA with the `SUPER` privilege can change the global value at runtime:

```
SET GLOBAL sort_buffer_size = 1024 * 256;
```

Each client that connects subsequently has its session variable initialized to the same value, but can change the value as it likes. A client that performs large sorts might increase the value:

```
SET SESSION sort_buffer_size = 1024 * 1024;
```

A `SET` statement that includes no `GLOBAL` or `SESSION` modifier changes the session value, if there is one.

There is alternative syntax for writing system variable references:

```
SET @@GLOBAL.var_name = value;
SET @@SESSION.var_name = value;
```

The `@@` syntax is more flexible. It can be used in statements other than `SET`, enabling you to retrieve or examine individual system variables:

```
mysql> SELECT @@GLOBAL.secure_auth, @@SESSION.sql_mode;
+----------------------+--------------------+
| @@GLOBAL.secure_auth | @@SESSION.sql_mode |
+----------------------+--------------------+
|                    1 | STRICT_ALL_TABLES  |
+----------------------+--------------------+
```

References to system variables using `@@` syntax with no `GLOBAL.` or `SESSION.` modifier access the session value if there is one, or the global value otherwise.

Other ways to access system variables include the `SHOW VARIABLES` statement and selecting from the `INFORMATION_SCHEMA GLOBAL_VARIABLES` and `SESSION_VARIABLES` tables.

If a setting exists only as a command option with no corresponding system variable, you cannot check its value at runtime. Fortunately, such options are rare. Nowadays, most new settings are created as system variables that can be examined at runtime.

22.2. Managing the Plug-In Interface

Problem

You want to exploit the capabilities offered by certain server plug-ins.

Solution

Learn how to control the plug-in interface.

Discussion

MySQL supports the use of plug-ins that extend server capabilities. There are plug-ins that implement storage engines, authentication methods, password policy, `INFORMATION_SCHEMA` tables, and more. The server enables you to specify which plug-ins to use, so that you can load just those you want, with no memory or processing overhead incurred for plug-ins you don't want.

This section provides the general background on controlling which plug-ins the server loads. Discussion elsewhere describes specific plug-ins and what they can do for you, including the authentication plug-ins (see Recipe 23.1), and `validate_password` (see Recipes 23.3 and 23.4).

The examples here refer to plug-in files using the *.so* ("shared object") filename suffix. If the suffix differs on your system, adjust the names accordingly (for example, use *.dll* on Windows). If you don't know the name of a given plug-in file, look in the directory named by the `plugin_dir` system variable, which is where the server expects to find plug-in files. For example:

```
mysql> SELECT @@plugin_dir;
+------------------------------+
| @@plugin_dir                 |
+------------------------------+
| /usr/local/mysql/lib/plugin/ |
+------------------------------+
```

To see which plug-ins are installed, use `SHOW PLUGINS` or query the `INFORMATION_SCHEMA PLUGINS` table.

Some plug-ins are built in, need not be enabled explicitly, and cannot be disabled. The `mysql_native_password` and `sha256_password` authentication plug-ins fall into this category.

Plug-in control at server startup

To install a plug-in only for a given server invocation, use the `--plugin-load-add` option at server startup, naming the file that contains the plug-in. To name multiple plug-ins as the option value, separate them with semicolons. Alternatively, use the option multiple times, with each instance naming a single plug-in. That makes it easy to enable or disable individual plug-ins by using the # character to selectively comment the corresponding lines:

```
[mysqld]
plugin-load-add=validate_password.so
plugin-load-add=adt_null.so
#plugin-load-add=semisync_master.so
#plugin-load-add=semisync_slave.so
```

The `--plugin-load-add` option was introduced in MySQL 5.6. In MySQL 5.5, you must use a single `--plugin-load` option that names all the plug-ins to be loaded in a semicolon-separated list:

```
[mysqld]
plugin-load=validate_password.so;adt_null.so
```

Clearly, for dealing with more than one plug-in, `--plugin-load-add` is superior for ease of administration.

Plug-in control at runtime

To install a plugin at runtime and make it persistent, use `INSTALL PLUGIN`. The server loads the plug-in (which becomes available immediately) and registers it in the `mysql.plugin` system table to cause it to load automatically for subsequent restarts. For example:

```
INSTALL PLUGIN validate_password SONAME 'validate_password.so';
```

The SONAME ("shared object name") clause specifies the file that contains the plug-in.

To disable a plug-in at runtime, use UNINSTALL PLUGIN. The server unloads the plug-in and removes its registration from the mysql.plugin table:

```
UNINSTALL PLUGIN validate_password;
```

INSTALL PLUGIN and UNINSTALL PLUGIN require the INSERT and DELETE privilege, respectively, for the mysql.plugin table.

22.3. Controlling Server Logging

Problem

You want to take advantage of log information the server can provide.

Solution

Learn the server options that control logging.

Discussion

The MySQL server can produce several logs:

The error log
: The error log contains information about problems or exceptional conditions the server encounters. This is useful information for debugging. In particular, if the server exits, check the error log for the reason. For example, if an exit occurs immediately after startup, it's likely that some setting in the server option file is misspelled or was set to an invalid value. The error log will contain a message to that effect.

The general query log
: The general query log indicates when each client connected and disconnected and what SQL statements it executed. This tells you how much and what activity each client is engaged in.

The slow query log
: The slow query log records statements that took a long time to execute (see the *MySQL Reference Manual* for the meaning of "a long time" because it can be influenced by several options). Queries that appear repeatedly in this log may be bottlenecks worth investigating to see whether they can be made more efficient.

The binary log
: The binary log contains a record of data changes made by the server. To set up replication, you must enable the binary log on the master server: it serves as the

storage medium for changes to be sent to slave servers. The binary log is also used, together with backup files, during data recovery operations.

Each log serves a different purpose and most can be turned on at your discretion, enabling you to use those that suit your administrative requirements. Each log can be written to a file, and some can be written to other destinations. The error log can be sent to your terminal or to the `syslog` facility. The general and slow query logs can be written to a file, to a table in the `mysql` database, or both.

To control server logging, add lines to your server option file that specify the desired types of logging. (Some settings can also be changed at runtime, as indicated later.) For example, the following lines in a server option file send the error log to the *err.log* file in the data directory, enable writing the general query and slow query logs to tables in the `mysql` database, and enable writing the binary log to the */var/mysql-logs* directory using files having names beginning with *binlog*:

```
[mysqld]
log_error=err.log
log_output=TABLE
general_log=1
slow_query_log=1
log-bin=/var/mysql-logs/binlog
```

For filenames in options that produce log output to files, logfiles are written under the data directory unless specified using full pathnames. The usual reason to use full pathnames is to write logfiles to a filesystem different from the one containing the data directory, a useful technique for dividing disk space use and I/O activity among physical devices.

The rest of this section provides details specific to controlling individual logs. The examples show the lines to include in your server option file to produce specific logging behavior. For some ideas about using the logs for diagnostic or activity assessment purposes, see Recipe 22.6.

For any log that you enable, see also Recipes 22.4 and 22.5 for log maintenance techniques. Logs increase in size over time, so you'll want to have a plan for managing them.

The error log

The error log cannot be disabled, but you can control where it's written. By default, on Unix, the error output goes to your terminal on Unix or to *`host_name.err`* in the data directory if you start the server using *mysqld_safe*. On Windows the default is *`host_name.err`* in the data directory. To specify the error log filename, set the `log_error` system variable.

Examples:

- Write the error log to the *err.log* file in the data directory:

```
[mysqld]
log_error=err.log
```

- As of MySQL 5.7.2, you can influence the amount of error log output by setting the `log_error_verbosity` system variable. Permitted values range from 1 (errors only) to 3 (errors, warnings, notes; the default). To see errors only, do this:

```
[mysqld]
log_error=err.log
log_error_verbosity=1
```

- On Unix, if you start the server using *mysqld_safe*, it's possible to redirect the error log to the `syslog` facility:

```
[mysqld_safe]
syslog
```

The general query and slow query logs

Several system variables control the general query and slow query logs. Each variable can be set at server startup or changed at runtime:

- `log_output` controls the log destinations. The value is `FILE` (log to files, the default), `TABLE` (log to tables), `NONE` (disable logging), or a comma-separated combination of values, in any order. `NONE` overrides any other value. If the value is `NONE`, other settings for these logs have no effect. Destination control applies to the general query and slow query logs together; you cannot write one to a file and the other to a table.
- `general_log` and `slow_query_log` enable or disable the respective logs. By default, each log is disabled. If you enable either of them, the server writes the log to the destinations specified by `log_output`, unless that variable is `NONE`.
- `general_log_file` and `slow_query_log_file` specify log filenames. The default names are *`host_name`.log* and *`host_name`-slow.log*; however, these settings have no effect unless `log_output` specifies `FILE` logging.

Examples:

- Write the general query log to the *query.log* file in the data directory:

```
[mysqld]
log_output=FILE
general_log=1
general_log_file=query.log
```

- Write the general and slow query logs to tables in the `mysql` database (the table names are `general_log` and `slow_log` and cannot be changed):

```
[mysqld]
log_output=TABLE
general_log=1
slow_query_log=1
```

- Write the general query log to a file named *query.log* and to the `general_log` table:

```
[mysqld]
log_output=FILE,TABLE
general_log=1
general_log_file=query.log
```

The binary log

To enable the binary log, use the `--log-bin` option, optionally specifying the logfile basename as the option value. The default basename is *`host_name`-bin*. The value for this option is a basename because the server creates binary logfiles in numbered sequence, automatically adding to the basename suffixes of *.000001*, *.000002*, and so forth. The server advances to the next file in the sequence when it starts, when the logs are flushed, and when the current file reaches the maximum logfile size (controlled by the `max_binlog_size` system variable). To have the server expire logfiles for you, set the `expire_logs_days` system variable to the age in days at which files become eligible for removal.

Examples:

- Enable the binary log, writing numbered files in the data directory having names beginning with *`host_name`-bin*:

```
[mysqld]
log-bin
```

- Enable the binary log, writing numbered files in the data directory having names beginning with *binlog*. Additionally, expire logfiles after a week:

```
[mysqld]
log-bin=binlog
max_binlog_size=4G
expire_logs_days=7
```

22.4. Rotating or Expiring Logfiles

Problem

Files used for logging grow indefinitely unless managed.

Problem

Available strategies include rotating a logfile through a set of names and expiring files by age. But different strategies apply to different logs, so consider the log type before choosing a strategy.

Discussion

Logfile rotation is a technique that renames a logfile through a series of one or more names. This maintains the file for a certain number of rotations, at which point it reaches the end of the sequence and its contents are discarded by being overwritten. Rotation can be applied to the error log, general query log, or slow query log.

Logfile expiration removes files when they reach a certain age. This technique applies to the binary log.

Both log management methods rely on log flushing to make sure that the current logfile has been closed properly. When you flush the logs, the server closes and reopens whichever of the files it is writing. If you rename the error, general query, or slow query logfile first, the server closes the current file and reopens a new one using the original name; this is what enables rotation of the current file while the server runs. The server also closes the current binary logfile and opens a new one with the next number in the sequence.

To flush the server logs, execute a `FLUSH LOGS` statement or use the *mysqladmin* `flush-logs` command. (Log flushing requires the `RELOAD` privilege.) The following discussion shows maintenance operations as performed at the command line, so it uses *mysqladmin*. The examples use *mv* as the file renaming command, which is applicable on Unix. On Windows, use *rename* instead.

Rotating the error, general query, or slow query log

To maintain a single file in a log rotation, rename the current logfile and flush the logs. Suppose that the error logfile is named *err.log* in the data directory. To rotate it, change location to the data directory, then execute these commands:

```
% mv err.log err.log.old
% mysqladmin flush-logs
```

When you flush the logs, the server opens a new *err.log* file. You can remove *err.log.old* at your leisure. To maintain an archive copy, include it in your filesystem backups before removing it.

To maintain a set of multiple rotated files, it's convenient to use a sequence of numbered suffixes. For example, to maintain a set of three old general query logfiles, do this:

```
% mv query.log.2 query.log.3
% mv query.log.1 query.log.2
```

```
% mv query.log query.log.1
% mysqladmin flush-logs
```

The first few times you execute the command sequence, the initial commands are unneeded until the respective *query.log.N* files exist.

Successive executions of that command sequence rotate *query.log* through the names *query.log.1*, *query.log.2*, and *query.log.3*; then *query.log.3* is overwritten and its contents lost. To maintain an archive copy, include the rotated files in your filesystem backups before removing them.

Rotating the binary log

The server creates binary logfiles in numbered sequence. To expire them, you need only arrange that it removes files when they're old enough. Several factors affect how many files the server creates and maintains:

- The frequency of server restarts and log flushing operations: one new file is generated each time either of those occurs.
- The size to which files can grow: larger sizes lead to fewer files. To control this size, set the `max_binlog_size` system variable.
- How old files are permitted to become: longer expiration times lead to more files. To control this age, set the `expire_logs_days` system variable. The server makes expiration checks at server startup and when it opens a new binary logfile.

The following settings enable the binary log, set the maximum file size to 4GB, and expire files after four days:

```
[mysqld]
log-bin=binlog
max_binlog_size=4G
expire_logs_days=4
```

You can also remove binary logfiles manually with the `PURGE BINARY LOGS` statement. For example, to remove all files up to and including the one named `binlog.001028`, do this:

```
PURGE BINARY LOGS TO 'binlog.001028';
```

If your server is a replication master, don't be too aggressive about removing binary logfiles. No file should be removed until you're certain its contents have been completely transmitted to all slaves.

Automating logfile rotation

To make it easier to perform a rotation operation, put the commands that implement it in a file to create a shell script. To perform the rotation automatically, arrange to execute the script from a job scheduler such as *cron*. The script will need to access

connection parameters that enable it to connect to the server to flush the logs, using an account that has the RELOAD privilege. One strategy is to put the parameters in an option file and pass the file to *mysqladmin* using a `--defaults-file=`*`file_name`* option. For example:

```
#!/bin/sh
mv err.log err.log.old
mysqladmin --defaults-file=/usr/local/mysql/data/flush-opts.cnf flush-logs
```

22.5. Rotating Log Tables or Expiring Log Table Rows

Problem

Tables used for logging grow indefinitely unless managed.

Problem

Rotate the tables or expire rows within them.

Discussion

Recipe 22.4 discusses rotation and expiration of logfiles. Analogous techniques apply to log tables:

- To rotate a log table, rename it and open a new table with the original name.
- To expire log table contents, remove rows older than a certain age.

The examples here demonstrate how to implement these methods using the general query log table, `mysql.general_log`. The same methods apply to the slow query log table, `mysql.slow_log`, or to any other table containing rows that have a timestamp. Prime examples are the Apache log table in Recipe 20.14, and the session-storage tables in Chapter 21.

To employ log table rotation, create an empty copy of the original table to serve as the new table (see Recipe 4.1), then rename the original table and rename the new one to take its place:

```
DROP TABLE IF EXISTS mysql.general_log_old, mysql.general_log_new;
CREATE TABLE mysql.general_log_new LIKE mysql.general_log;
RENAME TABLE mysql.general_log TO mysql.general_log_old,
  mysql.general_log_new TO mysql.general_log;
```

To employ log row expiration, you can either empty the table completely or selectively:

- To empty a log table completely, truncate it:

```
TRUNCATE TABLE mysql.general_log;
```

- To expire a table selectively, removing only rows older than a given age, you must know the name of the column that indicates row-creation time:

```
DELETE FROM mysql.general_log WHERE event_time < NOW() - INTERVAL 1 WEEK;
```

For automatic expiration, the statements for any of the techniques just described can be executed within a scheduled event (see Recipe 9.8). For example:

```
CREATE EVENT expire_general_log
  ON SCHEDULE EVERY 1 WEEK
  DO DELETE FROM mysql.general_log
     WHERE event_time < NOW() - INTERVAL 1 WEEK;
```

22.6. Monitoring the MySQL Server

Problem

You want to check how the server is operating.

Solution

Let the server tell you about itself.

Discussion

As your MySQL server runs, you'll have questions about aspects of its operation or performance. Or maybe it's *not* running and you want to know why. Here are some example questions:

- Is the server running? If so, how long has it been up?
- Why does the server quit immediately after I start it?
- How many queries is my server processing?
- How many simultaneous connections does the server permit? Is it close to running out?
- Is my slave replication server communicating with the master, or has replication stopped?
- Are the key caches sized properly for efficient operation?

This section discusses what you can find out, and how, by surveying the types of information available and how to use that information to answer questions. The purpose is not so much to consider specific monitoring problems as to illustrate your options so you can begin to answer your own questions, whatever they are.

To answer a question like any of those just posed, do this:

1. Determine which of the available information sources pertain to the problem at hand.
2. Choose an approach for using the information: Are you asking a one-time question? If so, maybe a few interactive queries are sufficient. If you're trying to solve an issue that may recur or for which you need continuous monitoring, a program-oriented approach is better. Will a script written entirely in SQL do the job, or do you need to write a program that queries the server and performs additional manipulation of the information obtained? (This is typical for operations that cannot be done in pure SQL, that have special output formatting requirements, and so forth.) If a task must run periodically, maybe you need to set up a scheduled event or *cron* job. For browser display, write a web script.

Sources of monitoring information

To follow the procedure just outlined, consider what information sources are available so that you can evaluate which are applicable and how usable they are for particular questions:

- System variables tell you how the server is configured. (Recipe 22.1 covers how to check these values.)
- Status variables provide information about operations the server is performing, such as number of statements executed, number of disk accesses, memory use, or cache efficiency. Status information can help indicate when configuration changes are needed, such as increasing the size of a too-small buffer to improve performance, or decreasing the size of an underused resource to reduce the server's memory footprint.
- `SHOW` statements and tables in the `INFORMATION_SCHEMA` database provide information ranging from processes running in the server to active storage engines and plug-ins to system and status variables. In many cases, these two sources provide the same or similar information, but in different display formats. (For example, the `SHOW PLUGINS` statement and the `PLUGINS` table are related.) Familiarity with both sources helps you choose which is more usable in a given situation:
  - For interactive use, `SHOW` is often more convenient because it involves less typing than `INFORMATION_SCHEMA` queries. Compare these two statements, which produce the same result:

    ```
    SHOW GLOBAL STATUS LIKE 'Threads_connected';

    SELECT VARIABLE_VALUE FROM INFORMATION_SCHEMA.GLOBAL_STATUS
    WHERE VARIABLE_NAME = 'Threads_connected';
    ```

  - `INFORMATION_SCHEMA` queries use `SELECT`, which is more expressive than `SHOW` and can be used for highly specific or complex queries, including joins.

— SHOW output cannot be saved using only SQL. Should you require further processing of an INFORMATION_SCHEMA query result, you can use INSERT INTO … SELECT to save the results in a table for further analysis (see Recipe 4.2). To obtain an individual value, assign a scalar subquery result to a variable:

```
SET @queries = (SELECT VARIABLE_VALUE
FROM INFORMATION_SCHEMA.GLOBAL_STATUS
WHERE VARIABLE_NAME = 'Queries');
```

- Some storage engines make information available about themselves. InnoDB, for example, has its own system and status variables. It also provides its own INFORMATION_SCHEMA tables and a set of InnoDB Monitors. The INFORMATION_SCHEMA tables provide more structured information and thus are more amenable to analysis using SQL, if they contain the information you want. To see which InnoDB-related tables are available, use this statement:

  ```
  SHOW TABLES FROM INFORMATION_SCHEMA LIKE 'innodb%';
  ```

  The Monitors produce unstructured output. You can eyeball it, but for programmatic use, you must parse or extract the information somehow. In some cases, a simple *grep* command might suffice:

  ```
  % mysql -E -e "SHOW ENGINE INNODB STATUS" | grep "Free buffers"
  Free buffers       4733
  ```

- The Performance Schema is designed for monitoring and provides a wealth of measurements, from high-level information such as which clients are connected down to fine-grained information such as which locks a statement holds or which files it has open. To use the Performance Schema, it must be enabled. This is the default as of MySQL 5.6.6; to enable it explicitly at server startup, use this configuration setting:

  ```
  [mysqld]
  performance_schema=1
  ```

- Server logs provide several types of information. Here are some suggestions for using them:

  — The error log alerts you to serious problems the server encounters. It's most suited to visual inspection because messages can originate from anywhere in the server and there is no fixed format to aid programmatic analysis. It's often only the last part of the file that's of interest, anyway, because you typically check this file to find the reason for the most recent problems.

  — The general query log indicates what queries clients are running. It can aid assessing the nature of the server's workload.

  — The slow log contains queries that may be inefficient. It can help you find candidates for optimization.

The server is able to write the general query and slow query logs to files, tables, or both. Log tables facilitate analysis better than the files; they are more structured and hence subject to analysis using SQL statements. The contents are also easier to interpret. Each query row in the `general_log` table shows the user associated with it. With the logfile, users are named only on connection lines. To identify a user's queries, you must extract the connection ID from the connection line and look for subsequent query lines with the same ID.

In addition, log tables are managed by the CSV storage engine, so the table datafiles are written in comma-separated values format. Look in the *mysql* directory under the server's data directory for files named *general_log.CSV* and *slow_log.CSV*. You can process them with tools that read CSV files.

To get information from a log, it must be enabled (see Recipe 22.3 for instructions).

- The `EXPLAIN` statement can be useful for checking long-running queries. Although `EXPLAIN` is most often used to see execution plans for prospective queries, MySQL 5.7.2 and up has the capability of using `EXPLAIN` to examine queries currently executing in other sessions. If a query seems to be stuck, this may help you understand why. Use `SHOW PROCESSLIST` or the `INFORMATION_SCHEMA PROCESSLIST` table to determine the connection ID of the session running the problem query, then point `EXPLAIN` at it:

  ```
  EXPLAIN FOR CONNECTION connection_id;
  ```

  `EXPLAIN` can produce output in tabular or `JSON` format. The latter can be parsed and manipulated by standard JSON modules in your programming language of choice.

Using monitoring information

After considering the information sources available to you, think about which is best suited to the question you seek to answer. If multiple sources apply, the best choice may depend on the context in which you intend to use it. The preceding summary of sources includes some remarks about their suitability to different contexts. Let's see how that works out in practice, by revisiting the original monitoring questions given at the beginning of this discussion and considering how to go about answering them. Keep in mind that these examples are illustrative, not exhaustive. For most, you can make implementation choices other than those shown.

Is the server running? If so, how long has it been up? To tell whether the server is running, just try connecting to it. If the connection succeeds or you get an error that's from the server itself, the server is up. *mysqladmin* `ping` is a good choice here, for interactive use or from within shell scripts. This result indicates the server is running:

```
% mysqladmin ping
mysqld is alive
```

This connection attempt fails, but the server itself returns the second error message, so it's not down:

```
% mysqladmin -u baduser ping
mysqladmin: connect to server at '127.0.0.1' failed
error: 'Access denied for user 'baduser'@'localhost' (using password: YES)'
```

This result indicates complete connection failure; the server is down:

```
% mysqladmin ping
mysqladmin: connect to server at '127.0.0.1' failed
error: 'Can't connect to MySQL server on '127.0.0.1' (61)'
```

If the server is not up, check the error log to find out why.

If the server is up, its uptime (in seconds) can be determined multiple ways:

- Use *mysqadmin* `status`:

  ```
  % mysqladmin status
  Uptime: 22158655  Threads: 2  Questions: 65733141  Slow queries: 34
  Opens: 6570  Flush tables: 1  Open tables: 95  Queries per second
  avg: 2.966
  ```

  A disadvantage of this approach for programmatic use is that you must parse the output to extract the value of interest.

- Examine the `Uptime` status variable:

  ```
  SHOW GLOBAL STATUS LIKE 'Uptime';
  ```

A server not running is obviously cause for concern. But there may be issues even if it is running. If you frequently find that server uptime resets in the absence of scheduled restarts, something may be causing the server to exit, and you should investigate. Again, check the error log to see why.

Why does the server quit immediately after I start it? If the server stops shortly after you start it, a likely cause is misconfiguration in the server option file. The error log helps you here. But don't be misled by mere warnings, which do not signify that the server quit. For example, the following message means only that `explicit_defaults_for_timestamp` need be set to make the warning go away:

```
2014-02-23T14:35:17.085998Z 0 [Warning] TIMESTAMP with implicit
DEFAULT value is deprecated. Please use --explicit_defaults_for_timestamp
server option (see documentation for more details).
```

Instead, check for `[ERROR]` lines, such as this:

```
2014-03-01T03:36:48.756313Z 0 [ERROR] mysqld: Error while setting
value 'TRADITONAL' to 'sql_mode'
```

The problem here is that `TRADITIONAL` was spelled incorrectly; correct it and start the server again.

How many queries is my server processing? This question might be prompted by simple curiosity, or there might be a performance issue. Monitoring statement execution over time and summarizing the results can reveal patterns, such as a time of day or day of week when activity is especially heavy. Perhaps several report generators are configured to start at the same time. Staggering them will help your server by spreading the load.

To answer the "how many queries" question, use status variable information:

```
mysql> SHOW GLOBAL STATUS LIKE 'Queries';
+---------------+----------+
| Variable_name | Value    |
+---------------+----------+
| Queries       | 65743031 |
+---------------+----------+
```

That tells you the number of statements executed since server startup, but it's an absolute value. You'll likely find a rate more useful: get the `Uptime` value and determine the `Queries` / `Uptime` ratio for a rate in statements per second. For a rate in a different unit, adjust the expression accordingly. For example, this tells you statements per minute:

```
mysql> SHOW GLOBAL STATUS LIKE 'Uptime';
+---------------+----------+
| Variable_name | Value    |
+---------------+----------+
| Uptime        | 22164600 |
+---------------+----------+
mysql> SELECT (65743031 * 60) / 22164600 AS 'statements/minute';
+-------------------+
| statements/minute |
+-------------------+
|          177.9677 |
+-------------------+
```

In programmatic context, you might write a long-running application that probes the server periodically for the `Queries` and `Uptime` values, to determine a running display of statement-execution activity. To avoid reconnecting each time you issue the statements, ask the server for its session timeout period and probe it at intervals shorter than that value. To get the session timeout value (in seconds), use this statement:

```
SELECT @@wait_timeout;
```

The default value is 28,800 (8 hours). If it's configured to a value shorter than your desired probe interval, set it higher:

```
SET wait_timeout = seconds;
```

The "MySQL Uncertainty Principle"

Heisenberg's uncertainty principle for measurement of quantum phenomena has a MySQL analog. If you monitor MySQL's status to see how it changes over time, you might notice the curious effect that, for some of the indicators, each time you take a measurement, you change the value you're measuring! For example, to determine the number of statements the server has received, use this statement:

```
SHOW GLOBAL STATUS LIKE 'Queries'
```

However, that statement is itself a statement, so each time you issue it, you cause the `Queries` value to change. In effect, your performance assessment instrument contaminates its own measurements, something you might want to take into account.

The preceding discussion uses `Queries`, which indicates the total number of statements executed. Options for more fine-grained analysis are available:

- The server maintains a set of `Com_`*xxx* status variables that count executions of particular statements. For example, `Com_insert` and `Com_update` count `INSERT` and `UPDATE` statements, respectively.
- To find out *which* queries were executed, check the general query log. To discover which ones were slow, check the slow query log. As mentioned previously, log tables are easier to analyze than logfiles because you can apply SQL statements to them. For example, this query summarizes number of statements per user, most active users first:

  ```
  SELECT COUNT(*), user_host FROM mysql.general_log
  GROUP BY user_host ORDER BY COUNT(*) DESC;
  ```

  A similar query shows which statements appear most often in the slow query log:

  ```
  SELECT COUNT(*), sql_text FROM mysql.slow_log
  GROUP BY sql_text ORDER BY COUNT(*) DESC;
  ```

  For information about summary and statistical techniques, see Chapter 8 and Chapter 15. Recipe 20.14 shows log-analysis queries in a related context (Apache logging).

How many simultaneous connections does the server permit? Is it close to running out? It's often the case that a server function is assessed using a combination of configuration settings plus current operational status. Typically, the former comes from system variables, whereas the latter comes from status variables. Connection management is an example of this concept. The `max_connections` system variable indicates the maximum number of simultaneous connections the server permits, and the `Threads_connected` status variable shows how many clients are currently connected. If `Threads_connected`

is regularly close to the value of `max_connections`, you might need to bump up the value of the latter. If there is always a wide gap, you can decrease `max_connections` to reduce resource allocation.

Is my slave replication server communicating with the master, or has replication stopped? If your server is a replication slave, having replication stop is something you want to avoid. Determining *why* replication stopped may involve some investigation, but finding out that a problem has occurred is a good first step toward resolving it.

The `SHOW SLAVE STATUS` statement helps you here. Alternatively, in MySQL 5.7.2 and up, the Performance Schema provides a set of tables containing replication configuration and status information.

Executing `SHOW SLAVE STATUS` on the slave provides columns showing the status of the I/O and SQL threads, and columns containing error information:

```
mysql> SHOW SLAVE STATUS\G
…
 Slave_IO_Running: Connecting
Slave_SQL_Running: Yes
…
    Last_IO_Errno: 2003
    Last_IO_Error: error reconnecting to master
                   'account' - retry-time: 60
                   retries: 7
   Last_SQL_Errno: 0
   Last_SQL_Error:
…
```

In this case, the slave is having trouble connecting to the master. If the slave is failing to execute the statements received from the master, `Slave_SQL_Running` will be `No` and the SQL error columns will show the error.

For communication problems, the Performance Schema `replication_connection_status` table provides information similar to that just shown:

```
mysql> SELECT * FROM performance_schema.replication_connection_status\G
…
             SOURCE_UUID: a2813915-ae3d-11e0-a30e-0019d1911a72
               THREAD_ID: NULL
          SERVICE_STATE: CONNECTING
RECEIVED_TRANSACTION_SET:
       LAST_ERROR_NUMBER: 2003
      LAST_ERROR_MESSAGE: error reconnecting to master
                          'account' - retry-time: 60
                          retries: 7
    LAST_ERROR_TIMESTAMP: 2014-03-01 09:17:28
```

For statement execution status, check the `replication_execute_status` table.

Are the key caches sized properly for efficient operation? The InnoDB and MyISAM storage engines each have a key cache. They serve to improve performance of index key lookups, so it's critical that they operate well. The main configuration setting for each is the cache size, and the operational status indicators are the number of requests for keys from the cache and the number of disk reads to pull values into the cache.

To determine the cache sizes, check the relevant system variables:

```
mysql> SELECT @@innodb_buffer_pool_size, @@key_buffer_size;
+---------------------------+-------------------+
| @@innodb_buffer_pool_size | @@key_buffer_size |
+---------------------------+-------------------+
|                 134217728 |           8388608 |
+---------------------------+-------------------+
```

You can also use `SHOW VARIABLES` or the `INFORMATION_SCHEMA GLOBAL_VARIABLES` table. For example:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.GLOBAL_VARIABLES WHERE
    -> VARIABLE_NAME IN ('INNODB_BUFFER_POOL_SIZE','KEY_BUFFER_SIZE');
+-------------------------+----------------+
| VARIABLE_NAME           | VARIABLE_VALUE |
+-------------------------+----------------+
| KEY_BUFFER_SIZE         | 8388608        |
| INNODB_BUFFER_POOL_SIZE | 134217728      |
+-------------------------+----------------+
```

The efficiency measure that determines how well a key cache is operating is its hit rate: the rate at which key requests are satisfied from the cache without reading keys from disk. If a key is in the cache, it's a hit; if not, it's a miss. The following expression computes the hit rate, where *`reads`* and *`requests`* indicate the number of disk reads and number of requests, respectively:

```
1 - (reads / requests)
```

To apply the expression to InnoDB or MyISAM, plug in the appropriate status variables:

```
1 - (Innodb_buffer_pool_reads / Innodb_buffer_pool_read_requests)
1 - (Key_reads / Key_read_requests)
```

If there have been no read requests, the expressions involve a division-by-zero operation, so it's necessary to account for that.

Values close to 1 indicate a high hit rate, which means that the key cache is very efficient. Values close to 0 indicate a low hit rate. If the value is not close to 1, consider making the cache larger by increasing the appropriate system variable (`innodb_buffer_pool_size` or `key_buffer_size`).

Suppose that you want to access the hit rate for the two caches by executing a SQL script from the command line. Because the status variables are not simply displayed but are used in a calculation, we require their values in a form that permits that use. `@@` syntax

does not apply here because that works only for system variables. Nor can we capture the result from a SHOW statement using only SQL. However, we can obtain any individual status variable value as a scalar subquery result. The following SQL script uses that approach to fetch the relevant status variables into user-defined variables and compute the hit rates:

```
# hitrate.sql: Show InnoDB and MyISAM key cache hit rate statistics
USE INFORMATION_SCHEMA;
SET @reads = (SELECT VARIABLE_VALUE FROM GLOBAL_STATUS
              WHERE VARIABLE_NAME = 'INNODB_BUFFER_POOL_READS');
SET @requests = (SELECT VARIABLE_VALUE FROM GLOBAL_STATUS
                 WHERE VARIABLE_NAME = 'INNODB_BUFFER_POOL_READ_REQUESTS');
SET @hit_rate = TRUNCATE(IFNULL(1 - (@reads/@requests), 0), 4);
SELECT 'InnoDB key cache hit rate' AS Message,
       @reads, @requests, @hit_rate;
SET @reads = (SELECT VARIABLE_VALUE FROM GLOBAL_STATUS
              WHERE VARIABLE_NAME = 'KEY_READS');
SET @requests = (SELECT VARIABLE_VALUE FROM GLOBAL_STATUS
                 WHERE VARIABLE_NAME = 'KEY_READ_REQUESTS');
SET @hit_rate = TRUNCATE(IFNULL(1 - (@reads/@requests), 0), 4);
SELECT 'MyISAM key cache hit rate' AS Message,
       @reads, @requests, @hit_rate;
```

Invoke the script to determine the key cache hit rates on demand:

```
% mysql -t < hitrate.sql
+---------------------------+--------+-----------+-----------+
| Message                   | @reads | @requests | @hit_rate |
+---------------------------+--------+-----------+-----------+
| InnoDB key cache hit rate |   6280 |  70138276 |    0.9999 |
+---------------------------+--------+-----------+-----------+
+---------------------------+--------+-----------+-----------+
| Message                   | @reads | @requests | @hit_rate |
+---------------------------+--------+-----------+-----------+
| MyISAM key cache hit rate |  23269 |   8902674 |    0.9973 |
+---------------------------+--------+-----------+-----------+
```

It looks like both caches are large enough to operate well.

The *hitrate.sql* script does the job, but four queries to get four status variables seems inefficient. You can see from the example how cumbersome it is to fetch their values individually. Manipulating status variables using a programming API permits a more straightforward process:

- Retrieve the entire set of status variables with a single query and store the result in a data structure that associates variable names with their values. This structure suffices no matter how many variables you monitor.
- Furthermore, it's reasonable to suppose that if we have one task that requires status variable values, we'll have other such tasks in the future. So it makes sense to write a routine to extract this information. It's necessary to write the routine only once

because it can be put in a library file for use from any number of monitoring applications.

The required information can be obtained from either `SHOW STATUS` or the `GLOBAL_STATUS` table. However, when executing queries within a program and saving the results, we must account for differences between `SHOW` statements and selecting from `INFORMATION_SCHEMA` tables. The following queries retrieve similar information, but the column headings differ in lettercase and sometimes in name, and variable names differ in lettercase:

```
mysql> SHOW GLOBAL STATUS;
+-------------------------------------------+-------------+
| Variable_name                             | Value       |
+-------------------------------------------+-------------+
| Aborted_clients                           | 1           |
| Aborted_connects                          | 6           |
…
mysql> SELECT * FROM INFORMATION_SCHEMA.GLOBAL_STATUS;
+-------------------------------------------+----------------+
| VARIABLE_NAME                             | VARIABLE_VALUE |
+-------------------------------------------+----------------+
| ABORTED_CLIENTS                           | 1              |
| ABORTED_CONNECTS                          | 6              |
…
```

To enable applications to be agnostic with respect to whether the variable information comes from `SHOW` or `INFORMATION_SCHEMA`, force variable names to a consistent lettercase and use that case in expressions that reference the variables. It doesn't matter which lettercase you choose, as long as you use it consistently. The following discussion uses uppercase.

Here's a simple routine (in Ruby) that takes a database handle, fetches the status variables, and returns them as a hash of values keyed by names:

```
def get_status_variables(dbh)
  vars = {}
  query = "SELECT VARIABLE_NAME, VARIABLE_VALUE FROM
           INFORMATION_SCHEMA.GLOBAL_STATUS"
  dbh.select_all(query).each { |name, value| vars[name.upcase] = value }
  return vars
end
```

To get the information using a `SHOW` statement instead, replace the query with this one:

```
query = "SHOW GLOBAL STATUS"
```

The code applies the `upcase` method to the variable names. That way, no matter whether the routine uses `GLOBAL_STATUS` or `SHOW` to obtain the information, the resulting hash has elements accessed by uppercase variable names.

To calculate a hit rate, pass the variable hash and the names of the reads and requests variables to this routine:

```
def cache_hit_rate(vars,reads_name,requests_name)
  reads = vars[reads_name].to_f
  requests = vars[requests_name].to_f
  hit_rate = requests == 0 ? 0 : 1 - (reads/requests)
  printf "        Key reads: %12d (%s)\n", reads, reads_name
  printf "Key read requests: %12d (%s)\n", requests, requests_name
  printf "         Hit rate: %12.4f\n", hit_rate
end
```

Now we're all set. Call the routines that fetch status information and calculate the hit rates like this:

```
statvars = get_status_variables(dbh)
cache_hit_rate(statvars,
               "INNODB_BUFFER_POOL_READS",
               "INNODB_BUFFER_POOL_READ_REQUESTS")
cache_hit_rate(statvars,
               "KEY_READS",
               "KEY_READ_REQUESTS")
```

Run the script to see your server's hit rates:

```
% hitrate.rb
        Key reads:         6280 (INNODB_BUFFER_POOL_READS)
Key read requests:     70138276 (INNODB_BUFFER_POOL_READ_REQUESTS)
         Hit rate:       0.9999
        Key reads:        23269 (KEY_READS)
Key read requests:      8902674 (KEY_READ_REQUESTS)
         Hit rate:       0.9974
```

For tasks involving system variables, code similar to `get_status_variables()` suffices. This implementation uses the `GLOBAL_VARIABLES` table:

```
def get_system_variables(dbh)
  vars = {}
  query = "SELECT VARIABLE_NAME, VARIABLE_VALUE FROM
           INFORMATION_SCHEMA.GLOBAL_VARIABLES"
  dbh.select_all(query).each { |name, value| vars[name.upcase] = value }
  return vars
end
```

To use `SHOW` instead, replace the query with this one:

```
query = "SHOW GLOBAL VARIABLES"
```

22.7. Creating and Using Backups

Problem

You want to protect yourself against data loss.

Solution

Use *mysqldump* to back up your databases.

Discussion

The *mysqldump* program provides an easy way to back up database contents. This program is discussed elsewhere, but primarily as a means of copying individual files (see Recipe 4.6). For administrative purposes, you're likely more interested in backing up entire databases, including nontable objects such as stored programs. This section shows some simple techniques for backup and recovery.

The *mysqldump* commands shown here include the `--routines` and `--events` options so that dump output includes definitions for stored functions and procedures and scheduled events. (There is also a `--triggers` option, but it's not used here because *mysqldump* dumps triggers with their associated tables by default.) To omit stored routine or scheduled event definitions from dump output, omit the `--routines` or `--events` option. To omit trigger definitions, use `--skip-triggers`.

To back up a single database:

```
% mysqldump --routines --events db1 > dump.sql
```

To reload the dump file:

```
% mysql db1 < dump.sql
```

That command reloads the file into the database from which it was dumped (`db1`), thereby restoring it to its state at the time of the dump. To make a *copy* of the original database, specify the name of a different database. (Create the database first if it doesn't exist.)

To back up multiple databases:

```
% mysqldump --routines --events --databases db1 db2 db3 > dump.sql
```

Normally, *mysqldump* treats nonoption arguments following the first as table names. The `--databases` option causes *mysqldump* to treat all such arguments as database names. That option also causes dump output to include `CREATE DATABASE` and `USE` statements for each database. This causes the reload operation to create each database as necessary, and makes it the default database so the following contents reload into it.

To reload the dump file:

```
% mysql < dump.sql
```

In this case, no database name is needed on the command line (and in fact is ignored if given) due to the `USE` statements in the dump file.

To back up all databases:

```
% mysqldump --routines --events --all-databases > dump.sql
```

No database names are given with this command. *mysqldump* asks the server which databases exist and dumps them all.

To reload the dump file:

```
% mysql < dump.sql
```

An all-databases dump can take a long time to produce and reload. Using several single- or multiple-database dumps may be a better strategy because you can work with smaller parts of your installation.

CHAPTER 23
Security

23.0. Introduction

This chapter covers security-related topics:

- The `mysql.user` table that contains MySQL account information
- Statements for managing MySQL user accounts
- Password strength checking and policy
- Password expiration
- Finding and fixing insecure accounts
- Finding and removing anonymous accounts and accounts that permit connections from many hosts

If you like, you can skip over the initial section that describes the `mysql.user` table, but I think you'll find that reading it will help you better understand later sections, which often discuss how SQL operations map onto underlying changes in that table.

Scripts shown in this chapter are located in the *routines* directory of the `recipes` distribution.

Whether you use the MySQL 5.5, 5.6, or 5.7 release series, it is best to use a recent version within the series. Changes to the authentication system occur in early development versions that may produce results that differ from the descriptions here.

Many of the techniques shown here require administrative access, such as the ability to modify tables in the `mysql` system database or use statements that require the `SUPER` privilege. For this reason, to carry out the operations described here, connect to the server as `root` rather than as `cbuser`.

23.1. Understanding the mysql.user Table

MySQL stores user account information in tables in the `mysql` system database. The `user` table is the most important because it contains account names and credentials. To see its structure, use this statement:

```
SHOW CREATE TABLE mysql.user;
```

The `user` table columns that concern us here specify account names and authentication information:

- The `User` and `Host` columns identify the account. MySQL account names comprise a combination of username and hostname values. For example, in the `user` table row for a `'cbuser'@'localhost'` account, the `User` and `Host` column values are `cbuser` and `localhost`, respectively. For a `'myuser'@'myhost.example.com'` account, those columns are `myuser` and `myhost.example.com`.
- The `plugin`, `Password`, and `authentication_string` columns store authentication credentials. MySQL does not store literal passwords in the `user` sytem table because that is insecure. Instead, the server computes a hash value from the password and stores the hash string.
  - The `plugin` column indicates which authentication plugin the server uses to check credentials for clients that attempt to use the account. Different plug-ins implement password hashing methods of varying encryption strength. The following table shows the plug-ins this chapter discusses:

Plug-in	Authentication method
`mysql_native_password`	Native password hashing
`mysql_old_password`	"Old" native password hashing (deprecated)
`sha256_password`	SHA-256 password hashing (MySQL 5.6.6 or later)

    MySQL Enterprise, the commercial version of MySQL, includes additional plug-ins for authenticating using PAM or Windows credentials. These enable use of passwords external to MySQL, such as Unix login passwords or native Windows services.
  - The `Password` column is used if the `plugin` column is `mysql_native_password` or `mysql_old_password`. An empty `Password` value means "no password,"

which is insecure. A nonempty value represents a hashed password in the format required by the respective plug-in.

— The `authentication_string` column is for use by plug-ins that do not use the `Password` column. For example, `sha256_password` uses `authentication_string` to store SHA-256 password hash values, which are cryptographically superior to native hashing but too long for the `Password` column.

Before MySQL 5.7.2, the server permits the `plugin` value to be empty. As of MySQL 5.7.2, the `plugin` column *must* be nonempty and the server disables any empty-plug-in account until a nonempty plug-in is assigned. However, even before 5.7.2, it's best if every account has a nonempty value:

- If the `plugin` column for an account is empty, the server authenticates clients using either `mysql_native_password` or `mysql_old_password` implicitly, making the choice based on the format of the hash value stored in the `Password` column. A nonempty plug-in makes the authentication method explicit.
- If you're running a version older than 5.7, modifying all accounts now to have a nonempty `plugin` value helps avoid issues when you upgrade to 5.7.

If your `user` table contains accounts that have an empty `plugin` value or use the deprecated `mysql_old_password` plug-in, you can fix them. Recipe 23.8 provides upgrade instructions.

23.2. Managing User Accounts

Problem

You are responsible for setting up accounts on your MySQL server.

Solution

Learn to use the account-management SQL statements.

Discussion

It's possible to modify the grant tables in the `mysql` database directly with SQL statements such as `INSERT` or `UPDATE`, but the MySQL account-management statements are more convenient. This section describes their use and covers these topics:

- Creating accounts (`CREATE USER`, `SET PASSWORD`)
- Assigning and checking privileges (`GRANT`, `REVOKE`, `SHOW GRANTS`)

- Removing and renaming accounts (`DROP USER`, `RENAME USER`)

Creating accounts

To create an account, use the `CREATE USER` statement, which creates a row in the `mysql.user` table. But before you do so, decide these three things:

- The account name, expressed in '*user_name*'@'*host_name*' format naming the user and the host from which the user will connect
- The account password
- The authentication plug-in the server should execute when clients attempt to use the account

Authentication plug-ins use hashing to encrypt passwords for storage and transmission. MySQL has several built-in plug-ins from which to choose:

- `mysql_native_password` implements the default password hashing method.
- `mysql_old_password` is similar but uses a hashing method that is less secure and is now deprecated. Avoid choosing this plug-in for new accounts. If your server has existing accounts that use it, Recipe 23.8 discusses how to identify and modify them to use `mysql_native_password` instead.
- `sha256_password` authenticates using SHA-256 password hash values, which are cryptographically more secure than hashes generated by `mysql_native_password`. This plug-in is available as of MySQL 5.6.6. It provides security beyond that afforded by `mysql_native_password`, but additional setup is required to use it. (Clients must connect using SSL or provide an RSA certificate.)

The `CREATE USER` statement is commonly used in one of these forms:

```
CREATE USER 'user_name'@'host_name' IDENTIFIED BY 'password';
CREATE USER 'user_name'@'host_name' IDENTIFIED WITH 'auth_plugin';
```

The first syntax creates the account and sets its password with a single statement. It also assigns an authentication plug-in implicitly, sort of:

- Before MySQL 5.6.6, the statement leaves the `plugin` column empty in the `user` table row for the account. It's preferable for the plug-in to be nonempty, for reasons discussed in Recipe 23.1.
- As of 5.6.6, the statement assigns the plug-in named by the `--default-authentication-plugin` setting (which is `mysql_native_password`, unless you change it at server startup).

With the second syntax, you must set the password separately using a subsequent SET PASSWORD statement, but because you specify the plug-in explicitly, it's always clear which one the user table row for the account will contain.

To create an account in a way that works consistently for any version of MySQL from 5.5 or later to ensure a designated nonempty plugin value, use this approach:

1. Create the account using a CREATE USER statement that names the authentication plug-in explicitly. Also, set the old_passwords system variable to select the password hashing method appropriate for the plug-in (this affects the PASSWORD() function in the next step). The following sequences show how to do this for each plug-in:

```
CREATE USER 'user_name'@'host_name' IDENTIFIED WITH 'mysql_native_password';
SET old_passwords = 0;

CREATE USER 'user_name'@'host_name' IDENTIFIED WITH 'mysql_old_password';
SET old_passwords = 1;

CREATE USER 'user_name'@'host_name' IDENTIFIED WITH 'sha256_password';
SET old_passwords = 2;
```

2. Set the account password:

```
SET PASSWORD FOR 'user_name'@'host_name' = PASSWORD('password');
```

   The PASSWORD() function hashes the password according to the old_passwords value just specified.

To assign privileges to the new account, which has none initially, use the GRANT statement described later in this section.

CREATE USER fails if the account already exists.

Writing an account-creation helper procedure

To make it easier to create new accounts, we can write a helper stored procedure named create_user() that does all the work, given the account username, hostname, password, and authentication plug-in:

- It issues the proper CREATE USER statement to specify the plug-in explicitly.
- It sets the password, or, if the password is given as NULL, leaves the password unset. (Presumably you'd specify NULL if you intend to assign the password later.)
- Before setting the password, it takes care of setting the old_passwords system variable to the appropriate value for the specified plug-in. It also saves and restores the current old_passwords value, to leave its value in your session undisturbed.
- To implement a policy that users must select their own password, it uses ALTER USER to expire the password immediately. The procedure skips this part if ALTER USER is

not available (the server is older than MySQL 5.6.7) or the account is for an anonymous user (who cannot set the account password to unexpire it). For more information about password expiration, see Recipe 23.5.

To use the procedure, invoke it like this:

```
CALL create_user('user_name','host_name','password','auth_plugin');
```

The procedure definition is shown following. It requires the helper routines `exec_stmt()` and `server_version()` from Recipes 9.9 and 10.9. Scripts to create these routines are located in the *routines* directory of the `recipes` distribution:

```
CREATE PROCEDURE create_user(user TEXT, host TEXT,
                             password TEXT, plugin TEXT)
BEGIN
  DECLARE account TEXT;
  SET account = CONCAT(QUOTE(user),'@',QUOTE(host));
  CALL exec_stmt(CONCAT('CREATE USER ',account,
                        ' IDENTIFIED WITH ',QUOTE(plugin)));
  IF password IS NOT NULL THEN
    BEGIN
      DECLARE saved_old_passwords INT;
      SET saved_old_passwords = @@old_passwords;
      CASE plugin
        WHEN 'mysql_native_password' THEN SET old_passwords = 0;
        WHEN 'mysql_old_password' THEN SET old_passwords = 1;
        WHEN 'sha256_password' THEN SET old_passwords = 2;
        ELSE SIGNAL SQLSTATE 'HY000'
                    SET MYSQL_ERRNO = 1525,
                        MESSAGE_TEXT = 'unhandled auth plugin';
      END CASE;
      CALL exec_stmt(CONCAT('SET PASSWORD FOR ',account,
                            ' = PASSWORD(',QUOTE(PASSWORD),')'));
      SET old_passwords = saved_old_passwords;
    END;
  END IF;
  IF server_version() >= 50607 AND user <> '' THEN
    CALL exec_stmt(CONCAT('ALTER USER ',account,' PASSWORD EXPIRE'));
  END IF;
END;
```

Assigning and checking privileges

Suppose that you have just created an account named `'user1'@'localhost'`. You can assign privileges to it with `GRANT`, remove privileges from it with `REVOKE`, and check its privileges with `SHOW GRANTS`.

`GRANT` has this syntax:

```
GRANT privileges ON scope TO account;
```

Here, *account* names the account to be granted the privileges, *privileges* indicates what they are, and *scope* indicates the privilege scope, or level at which they apply. The *privileges* value can be ALL (or ALL PRIVILEGES) to specify all privileges available at the given level, or a comma-separated list of one or more privilege names such as SELECT or CREATE. (For a full discussion of available privileges and GRANT syntax not shown here, see the *MySQL Reference Manual*.)

The following examples illustrate the syntax for granting privileges at each level.

- Granting privileges globally enables the account to perform administrative operations or operations on any database:

```
GRANT FILE ON *.* TO 'user1'@'localhost';
GRANT CREATE TEMPORARY TABLES, LOCK TABLES ON *.* TO 'user1'@'localhost';
```

- Granting privileges at the database level enables the account to perform operations on objects within the named database:

```
GRANT ALL ON cookbook.* TO 'user1'@'localhost';
```

- Granting privileges at the table level enables the account to perform operations on the named table:

```
GRANT SELECT ON mysql.user TO 'user1'@'localhost';
```

- Granting privileges at the column level enables the account to perform operations on the named table column:

```
GRANT SELECT(User,Host), UPDATE(password_expired)
ON mysql.user TO 'user1'@'localhost';
```

- Granting privileges at the procedure level enables the account to perform operations on the named stored procedure:

```
GRANT EXECUTE ON PROCEDURE cookbook.exec_stmt TO 'user1'@'localhost';
```

  Use FUNCTION rather than PROCEDURE if the routine is a stored function.

To verify the privilege assignments, use SHOW GRANTS:

```
mysql> SHOW GRANTS FOR 'user1'@'localhost';
+-------------------------------------------------------------------------+
| Grants for user1@localhost                                              |
+-------------------------------------------------------------------------+
| GRANT FILE, CREATE TEMPORARY TABLES, LOCK TABLES                        |
| ON *.* TO 'user1'@'localhost'                                           |
| GRANT ALL PRIVILEGES ON `cookbook`.* TO 'user1'@'localhost'             |
| GRANT SELECT, SELECT (User, Host), UPDATE (password_expired)            |
| ON `mysql`.`user` TO 'user1'@'localhost'                                |
| GRANT EXECUTE ON PROCEDURE `cookbook`.`exec_stmt` TO 'user1'@'localhost' |
+-------------------------------------------------------------------------+
```

To see your own privileges, omit the FOR clause.

REVOKE syntax is generally similar to GRANT but uses FROM rather than TO:

```
REVOKE privileges ON scope FROM account;
```

Thus, to remove the privileges just granted to 'user1'@'localhost', use these REVOKE statements (and SHOW GRANTS to verify that they were removed):

```
mysql> REVOKE FILE ON *.* FROM 'user1'@'localhost';
mysql> REVOKE CREATE TEMPORARY TABLES, LOCK TABLES
    -> ON *.* FROM 'user1'@'localhost';
mysql> REVOKE ALL ON cookbook.* FROM 'user1'@'localhost';
mysql> REVOKE SELECT ON mysql.user FROM 'user1'@'localhost';
mysql> REVOKE SELECT(User,Host), UPDATE(password_expired)
    -> ON mysql.user FROM 'user1'@'localhost';
mysql> REVOKE EXECUTE ON PROCEDURE cookbook.exec_stmt
    -> FROM 'user1'@'localhost';
mysql> SHOW GRANTS FOR 'user1'@'localhost';
+-----------------------------------------+
| Grants for user1@localhost              |
+-----------------------------------------+
| GRANT USAGE ON *.* TO 'user1'@'localhost' |
+-----------------------------------------+
```

Removing accounts

To get rid of an account, use the DROP USER statement:

```
DROP USER 'user1'@'localhost';
```

The statement removes all rows associated with the account in all grant tables; you need not use REVOKE to remove its privileges first. An error occurs if the account does not exist.

Renaming accounts

To change an account name, use RENAME USER, specifying the current and new names:

```
RENAME USER 'currentuser'@'localhost' TO 'newuser'@'localhost';
```

An error occurs if the current account does not exist or the new account already exists.

23.3. Implementing a Password Policy

Problem

You want to ensure that MySQL accounts do not use weak passwords.

Solution

Use the `validate_password` plug-in to implement a password policy. New passwords must satisfy the policy, whether those chosen by the DBA for new accounts or by existing users changing their password.

Discussion

This technique requires the `validate_password` plug-in to be enabled. For plug-in installation instructions, see Recipe 22.2.

When `validate_password` is enabled, it exposes a set of system variables that enable you to configure it. These are the default values:

```
mysql> SHOW VARIABLES LIKE 'validate_password%';
+--------------------------------------+--------+
| Variable_name                        | Value  |
+--------------------------------------+--------+
| validate_password_dictionary_file    |        |
| validate_password_length             | 8      |
| validate_password_mixed_case_count   | 1      |
| validate_password_number_count       | 1      |
| validate_password_policy             | MEDIUM |
| validate_password_special_char_count | 1      |
+--------------------------------------+--------+
```

Suppose that you want to implement a policy that enforces these requirements for passwords:

- At least 10 characters long
- Contains uppercase and lowercase characters
- Contains at least two digits
- Contains at least one special (nonalphanumeric) character

To put that policy in place, start the server with options that enable the plug-in and set the values of the system variables that configure the policy requirements. For example, put these lines in your server option file:

```
[mysqld]
plugin-load-add=validate_password.so
validate_password_length=10
validate_password_mixed_case_count=1
validate_password_number_count=2
validate_password_special_char_count=1
```

After starting the server, verify the settings:

```
mysql> SHOW VARIABLES LIKE 'validate_password%';
+--------------------------------------+--------+
| Variable_name                        | Value  |
+--------------------------------------+--------+
| validate_password_dictionary_file    |        |
| validate_password_length             | 10     |
| validate_password_mixed_case_count   | 1      |
| validate_password_number_count       | 2      |
| validate_password_policy             | MEDIUM |
| validate_password_special_char_count | 1      |
+--------------------------------------+--------+
```

Now the `validate_password` plug-in prevents assigning passwords too weak for the policy:

```
mysql> SET PASSWORD = PASSWORD('weak-password');
ERROR 1819 (HY000): Your password does not satisfy the current
policy requirements
mysql> SET PASSWORD = PASSWORD('Str0ng-Pa33w@rd');
Query OK, 0 rows affected (0.00 sec)
```

The preceding instructions leave the `validate_password_policy` system variable set to its default value (`MEDIUM`), but you can change it to control how the server tests passwords:

- `MEDIUM` enables tests for password length and the number of numeric, uppercase/lowercase, and special characters.
- To be less rigorous, set the policy to `LOW`, which enables only the length test. To also permit shorter passwords, decrease the required length (`validate_password_length`).
- To be more rigorous, set the policy to `STRONG`, which is like `MEDIUM` but also enables you to have passwords checked against a dictionary file, to prevent use of passwords that match any word in the file. Comparisons are not case sensitive.

  To use a dictionary file, set the value of `validate_password_dictionary_file` to the filename at server startup. The file should contain lowercase words, one per line. MySQL distributions include a *dictionary.txt* file in the *share* directory that you can use, and Unix systems often have a */usr/share/dict/words* file.

Putting a password policy in place has no effect on existing passwords. To require users to choose a new password that satisfies the policy, expire their current password (see Recipe 23.5).

23.4. Checking Password Strength

Problem

You want to assign or change a password but verify first that it's not weak.

Solution

Use the `VALIDATE_PASSWORD_STRENGTH()` function.

Discussion

The `validate_password` plug-in not only implements policy for new passwords, it provides a SQL function, `VALIDATE_PASSWORD_STRENGTH()`, that enables strength testing of prospective passwords. Uses for this function include:

- An administrator wants to check passwords to be assigned to new accounts.
- An individual user wants to choose a new password but seeks assurance in advance how strong it is.

To use `VALIDATE_PASSWORD_STRENGTH()`, the `validate_password` plug-in must be enabled. For plug-in installation instructions, see Recipe 22.2.

`VALIDATE_PASSWORD_STRENGTH()` returns a value from 0 (weak) to 100 (strong):

```
mysql> SELECT VALIDATE_PASSWORD_STRENGTH('abc') ;
+----------------------------------+
| VALIDATE_PASSWORD_STRENGTH('abc') |
+----------------------------------+
|                                0 |
+----------------------------------+
mysql> SELECT VALIDATE_PASSWORD_STRENGTH('weak-password');
+-------------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('weak-password') |
+-------------------------------------------+
|                                        50 |
+-------------------------------------------+
mysql> SELECT VALIDATE_PASSWORD_STRENGTH('Str0ng-Pa33w@rd');
+-----------------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('Str0ng-Pa33w@rd') |
+-----------------------------------------------+
|                                           100 |
+-----------------------------------------------+
```

23.5. Expiring Passwords

Problem

You want users to pick a new MySQL password.

Solution

The `ALTER USER` statement expires passwords.

Discussion

MySQL 5.6.7 and up provides an `ALTER USER` statement that enables an administrator to expire an account's password:

```
ALTER USER 'cbuser'@'localhost' PASSWORD EXPIRE;
```

Here are some uses for password expiration:

- You can implement a policy that new users must select a new password when first connecting: immediately expire the password for each new account you create.
- If you impose a stricter policy on acceptable passwords (see Recipe 23.3), you can expire all existing passwords to require each user to choose a new one that meets the more stringent requirements.

`ALTER USER` affects a single account. It works by setting the `password_expired` column to `Y` for the appropriate `mysql.user` row. To "cheat" and expire passwords for all non-anonymous accounts at once, do this (anonymous users cannot reset their password, so expiring those would be unfriendly):

```
UPDATE mysql.user SET password_expired = 'Y' WHERE User <> '';
FLUSH PRIVILEGES;
```

Alternatively, to affect all accounts but avoid modifying the grant tables directly, use a stored procedure that loops through all accounts and executes `ALTER USER` for each:

```
CREATE PROCEDURE expire_all_passwords()
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE account TEXT;
  DECLARE cur CURSOR FOR
    SELECT CONCAT(QUOTE(User),'@',QUOTE(Host)) AS account
    FROM mysql.user WHERE User <> '';
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur;
  expire_loop: LOOP
    FETCH cur INTO account;
    IF done THEN
```

```
      LEAVE expire_loop;
    END IF;
    CALL exec_stmt(CONCAT('ALTER USER ',account,' PASSWORD EXPIRE'));
  END LOOP;
  CLOSE cur;
END;
```

The procedure requires the `exec_stmt()` helper routine (see Recipe 9.9). Scripts to create these routines are located in the *routines* directory of the `recipes` distribution.

23.6. Assigning Yourself a New Password

Problem

You want to change your password.

Solution

Use the `SET PASSWORD` statement.

Discussion

To assign yourself a new password, use the `SET PASSWORD` statement and the `PASSWORD()` function:

```
SET PASSWORD = PASSWORD('my-new-password');
```

`SET PASSWORD` permits a `FOR` clause that enables you to specify which account gets the new password:

```
SET PASSWORD FOR 'user_name'@'host_name' = PASSWORD('my-new-password');
```

This latter syntax is primarily for DBAs because it requires the `UPDATE` privilege for the `mysql` database.

If `SET PASSWORD` complains about the password hash being in the wrong format, try again after setting `old_passwords` to select the hashing method appropriate for the authentication plug-in associated with your account. Recipe 23.2 provides these values.

To check the strength of a password you're considering, use the `VALIDATE_PASSWORD_STRENGTH()` function (see Recipe 23.4).

23.7. Resetting an Expired Password

Problem

You cannot use MySQL because your DBA expired your password.

Solution

Assign yourself a new password.

Discussion

If the MySQL administrator has expired your password, MySQL will let you connect, but not do much of anything else:

```
% mysql --user=cbuser --password
Enter password: ******
mysql> SELECT CURRENT_USER();
ERROR 1820 (HY000): You must SET PASSWORD before executing this statement
```

If you see that message, reset your password so that you can work normally again:

```
mysql> SET PASSWORD = PASSWORD('my-new-password');
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT CURRENT_USER(); -- now you can work again
+------------------+
| CURRENT_USER()   |
+------------------+
| cbuser@localhost |
+------------------+
1 row in set (0.00 sec)
```

Technically, MySQL does not require a *new* password to replace an expired password, so you can assign yourself your current password to unexpire it. The exception is that if the password policy has become more restrictive and your current password no longer satisfies it, a stronger password must be chosen.

For more information about changing your password, see Recipe 23.6.

23.8. Finding and Fixing Insecure Accounts

Problem

Your MySQL installation includes accounts that have no password or use deprecated and insecure password hashing.

Solution

Upgrade those accounts to use a better password hashing method.

Discussion

Security is important and MySQL has improved user account security over time. An early change occurred way back in MySQL 4.1, with the introduction of a better pass-

word hashing method than the original pre-4.1 method. (MySQL does not store literal passwords in the `mysql.user` system table because that is insecure. Instead, the server computes a hash value from the password and stores the hash string.) More recent authentication changes include the introduction in MySQL 5.6 of the `sha256_password` plug-in that implements SHA-256 password hashing and the `validate_password` plug-in that implements password policy and password strength assessment. This section describes characteristics of the 4.1 and (less secure) original hashing methods and shows how to upgrade accounts that use the original method so they use the 4.1 method instead. For information about the `sha256_password` and `validate_password` plug-ins, see Recipes 23.2 and 23.4.

For any account with a nonempty `Password` value in its `user` table row, you can tell which hashing method generated it:

- The hashing method introduced in MySQL 4.1 produces 41-character hash values beginning with a `*` character. This is the "4.1" or "native" hashing method. For accounts that have this type of password hash, the server authenticates connection attempts using the `mysql_native_password` plug-in.
- The original hashing method produces 16-character hash values. This is the "pre-4.1" or "old" hashing method. The server authenticates accounts that have this type of password hash using the `mysql_old_password` authentication plug-in.

To see the difference between the two hash formats, generate hash values explicitly:

```
mysql> SET old_passwords = 0;
mysql> SELECT OLD_PASSWORD('mypass') AS old, PASSWORD('mypass') AS new\G
*************************** 1. row ***************************
old: 6f8c114b58f2ce9e
new: *6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4
```

The example sets `old_passwords` to 0 because `PASSWORD()` uses the pre-4.1 hashing method and returns the same result as `OLD_PASSWORD()` if `old_passwords` is set to 1.

Administrators should avoid creating accounts that use the older, less secure pre-4.1 hashing method. If your MySQL installation has accounts that have old password hashes, you can upgrade them to use the 4.1 hashing method. (This will become necessary eventually, anyway. Pre-4.1 hashing is deprecated as of MySQL 5.6 and support for it will be dropped at some point.)

Additionally, each account should have a nonempty password.

To identify and upgrade insecure accounts, use this procedure:

1. Determine whether your `user` table contains accounts with weak security. A "weak" account has either of these characteristics:

- The `plugin` column is `mysql_native_password` but the `Password` column is empty.
- The `plugin` column is empty or `mysql_old_password`. (If the value is empty, the server authenticates clients using either `mysql_native_password` or `mysql_old_password`, making the choice based on the hash format of the value stored in the `Password` column. To prevent the possibility of implicit authentication using `mysql_old_password`, set the plug-in to `mysql_native_password`.)

Use this query to find weak accounts with those characteristics:

```
SELECT User, Host, plugin, Password FROM mysql.user
WHERE (plugin = 'mysql_native_password' AND Password = '')
      OR plugin IN ('','mysql_old_password');
```

2. Before upgrading a weak account, consider whether the account is even necessary. Perhaps it was created long ago for a project that's no longer used and you can simply remove it:

```
DROP USER 'olduser'@'localhost';
```

   The result is one less account to be protected and one less point of exploit.

3. If a weak account must be retained, upgrade it:

   - If the plug-in is empty or `mysql_old_password`, change it to `mysql_native_password` so that pre-4.1 password hashing cannot be used.
   - If the password is empty or in pre-4.1 hash format, assign a new password using 4.1 hashing.

Suppose that a server's user population includes accounts with the following authentication characteristics, most of which need improvement. (All have a `Host` value of `localhost`, although it's not shown here.)

```
mysql> SELECT User, plugin, Password FROM mysql.user
    -> WHERE User LIKE 'user%' AND Host = 'localhost' ORDER BY User;
+-------+-----------------------+-------------------------------------------+
| User  | plugin                | Password                                  |
+-------+-----------------------+-------------------------------------------+
| user1 | mysql_native_password | *6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4 |
| user2 |                       |                                           |
| user3 | mysql_old_password    |                                           |
| user4 |                       | 6f8c114b58f2ce9e                          |
| user5 | mysql_old_password    | 6f8c114b58f2ce9e                          |
| user6 | mysql_native_password |                                           |
| user7 |                       | *6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4 |
+-------+-----------------------+-------------------------------------------+
```

The requirements for better security are that each account names the `mysql_native_password` plug-in explicitly and has a nonempty password in 4.1 hash format.

Measured against those requirements, only the `user1` account has acceptable values. (It's the only account not selected by the "identify weak accounts" query shown earlier.) Each of the other accounts is deficient in some way. The following instructions describe how to address their weaknesses.

In general, it's preferable to manipulate MySQL accounts using SQL statements intended for that purpose, such as `CREATE USER` or `SET PASSWORD`, and to avoid modifying the grant tables directly using statements such as `INSERT` or `UPDATE`. But some operations are more straightforward using direct manipulation (and sometimes not possible to perform otherwise), so the following instructions include some direct modifications of the `user` table, even though that goes against convention. A consequence of direct manipulation is that `FLUSH PRIVILEGES` is required following `UPDATE`, to ensure that the server refreshes the account information it caches in memory.

For each account for which you reassign the password, you must either know the current password or assign a temporary password. In the latter case, contact the account owner, provide the temporary password, and ask the owner to choose a new one.

Begin by setting the `old_passwords` system variable to 0, to ensure that `PASSWORD()` uses the 4.1 hashing method, not the pre-4.1 method:

```
SET old_passwords = 0;
```

That done, upgrade each account per its particular weaknesses. Note that the `UPDATE` statements specify both `User` and `Host` (not just `User`) to uniquely identify the single account to update:

- `user1` weaknesses: None. The account specifies the native plug-in explicitly and the password is nonempty in 4.1 hash format. Actions: None needed.
- `user2` through `user5` have different weaknesses, but in each case the statements to implement the required security upgrade are the same:
  - `user2` weaknesses: No plug-in named; password is empty. Actions: Specify the native plug-in; assign a password.
  - `user3` weaknesses: Uses the old plug-in; password is empty. Actions: Change to the native plug-in; assign a password.
  - `user4` weaknesses: No plug-in named; password uses pre-4.1 hash. Actions: Specify the native plug-in; upgrade password to 4.1 hash.
  - `user5` weaknesses: Uses the old plug-in; password uses pre-4.1 hash. Actions: Change to the native plug-in; upgrade password to 4.1 hash.

  To address the issues for any of `user2` through `user5`, use the following statements (substituting the proper username for `user2` as necessary):

```
UPDATE mysql.user
SET plugin = 'mysql_native_password', Password = PASSWORD('mypass')
WHERE User = 'user2' AND Host = 'localhost';
FLUSH PRIVILEGES;
```

- `user6` weakness: Password is empty. Action: Assign a password.

```
SET PASSWORD FOR 'user6'@'localhost' = PASSWORD('mypass');
```

- `user7` weakness: No plug-in named. Action: Specify the native plug-in.

```
UPDATE mysql.user
SET plugin = 'mysql_native_password'
WHERE User = 'user7' AND Host = 'localhost';
FLUSH PRIVILEGES;
```

23.9. Disabling Use of Accounts with Pre-4.1 Passwords

Problem

The original pre-4.1 hashing method is less secure than other methods and you want to prevent accounts from using it.

Solution

Set the `secure_auth` system variable to prevent such accounts from connecting to the server. To be more user friendly, upgrade affected accounts first.

Discussion

The hashing method used by the `mysql_old_password` authentication plug-in is not as secure as the method used by `mysql_native_password`. In addition, `mysql_old_password` is deprecated and eventually will no longer be supported. To prevent its use and prepare for the day when support for it ceases, take these steps:

1. Identify accounts that use `mysql_old_password` and upgrade them to use `mysql_native_password` (see Recipe 23.8). Do this first so as not to lock out accounts in the next step.
2. Start the server with the `secure_auth` system variable enabled. That's been the default value since MySQL 5.6.5, but you can check whether your server's setting differs:

```
mysql> SELECT @@secure_auth;
+---------------+
| @@secure_auth |
+---------------+
|             0 |
+---------------+
```

If the value is 0, enable the variable by starting the server with the value set to 1. For example, use these lines in an option file:

```
[mysqld]
secure_auth=1
```

At this point, accounts that use pre-4.1 password hashes can no longer connect.

23.10. Finding and Removing Anonymous Accounts

Problem

You want to ensure that your MySQL server can be used only by accounts associated with specific usernames.

Solution

Identify and remove anonymous accounts.

Discussion

An "anonymous" account is one that has an empty user part in the account name, such as `''@'localhost'`. An empty user matches any name because the purpose of an anonymous account is to permit anyone who knows its password to connect from the named host (`localhost` in this case). This is a convenience because the DBA need not set up individual accounts for separate users. But there are security implications as well:

- Such accounts often are given no password, enabling their use with no authentication at all.
- You cannot associate database activity with specific users (for example, by checking the server query log or examining `SHOW PROCESSLIST` output), making it more difficult to tell who is doing what.

If the preceding points persuade you that anonymous accounts are not a good thing, use the following instructions to identify and remove them:

1. The `User` column is empty in the `mysql.user` rows for anonymous accounts, so you can identify them like this:

```
mysql> SELECT User, Host FROM mysql.user WHERE User = '';
+------+---------------+
| User | Host          |
+------+---------------+
|      | %.example.com |
|      | localhost     |
+------+---------------+
```

2. The SELECT output shows two anonymous accounts. Remove each using a DROP USER statement with the corresponding account name:

```
mysql> DROP USER ''@'localhost';
mysql> DROP USER ''@'%.example.com';
```

23.11. Modifying "Any Host" and "Many Host" Accounts

Problem

You want to ensure that MySQL accounts cannot be used from an overly broad set of hosts.

Solution

Find and fix accounts containing % or _ in the host part.

Discussion

The host part of MySQL account names can contain the SQL pattern characters % and _ (see Recipe 5.8). These names match client connection attempts from any host that matches the pattern. For example, the account `'user1'@'%'` permits `user1` to connect from any host whatsoever, and `'user2'@'%.example.com'` permits `user2` to connect from any host in the `example.com` domain.

Patterns in the host part of account names provide a convenience that enables a DBA to create an account that permits connections from multiple hosts. They correspondingly increase security risks by increasing the number of hosts from which intruders can attempt to connect. If you consider this a concern, identify the accounts and either remove them or change the host part to be more specific.

There are several ways to find accounts with % or _ in the host part. Here are two:

```
WHERE Host LIKE '%\%%' OR Host LIKE '%\_%';
WHERE Host REGEXP '[%_]';
```

The LIKE expression is more complex because we must look for each pattern character separately and escape it to search for literal instances. The REGEXP expression requires no escaping because those characters are not special in regular expressions, and a character class permits both to be found with a single pattern. So let's use that expression:

1. Identify pattern-host accounts in the `mysql.user` table like this:

```
mysql> SELECT User, Host FROM mysql.user WHERE Host REGEXP '[%_]';
+-------+---------------+
| User  | Host          |
+-------+---------------+
```

```
| user1 | %            |
| user2 | %.example.com |
| user3 | _.example.com |
+-------+--------------+
```

2. To remove an identified account, use `DROP USER`:

```
mysql> DROP USER 'user1'@'%';
mysql> DROP USER 'user3'@'_.example.com';
```

   Alternatively, rename an account to make the host part more specific:

```
mysql> RENAME USER 'user2'@'%.example.com' TO 'user2'@'host17.example.com';
```

Index

Symbols

A

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

Paul DuBois is one of the primary contributors to the *MySQL Reference Manual*, a renowned online manual that has supported MySQL administrators and database developers for years. He is a member of the MySQL documentation team at Oracle Corporation and is also the author of *MySQL* (Addison-Wesley Professional); *MySQL Cookbook*; *Using csh & tcsh*; *Software Portability with imake*; and *MySQL and Perl for the Web* (New Riders).

Colophon

The animal on the cover of *MySQL Cookbook, Third Edition* is a green anole. These common lizards can be found in the southeastern United States, the Caribbean, and South America. Green anoles dwell in moist, shady enivornments, such as inside trees and shrubs. They subsist on small insects like crickets, roaches, moths, grubs, and spiders.

Green anoles are slight in build, with narrow heads and long, slender tails that can be twice as long as their bodies. The special padding on their feet enables them to climb, cling to, and run on any surface. They range in size from six to eight inches long. Though, as their name implies, green anoles are usually bright green, their color can change to match their surroundings, varying among gray-brown, brown, and green. Male anoles have pink dewlaps that they extend when courting or protecting their territory.

The cover image is from the Dover Pictorial Archive. The cover fonts are URW Typewriter 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.