Paul DuBois

Fifth Edition

MySQL®

Developer's Library

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

MySQL

Fifth Edition

Developer's Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

PHP & MySQL Web Development
Luke Welling & Laura Thomson
ISBN 978-0-672-32916-6

MySQL
Paul DuBois
ISBN-13: 978-0-321-83387-7

Linux Kernel Development
Robert Love
ISBN-13: 978-0-672-32946-3

Python Essential Reference
David Beazley
ISBN-13: 978-0-672-32978-4

PostgreSQL
Korry Douglas
ISBN-13: 978-0-672-32756-8

C++ Primer Plus
Stephen Prata
ISBN-13: 978-0-321-77640-2

Developer's Library books are available in print and in electronic formats at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

Developer's Library
informit.com/devlibrary

MySQL

Fifth Edition

Paul DuBois

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

MySQL, Fifth Edition

ISBN-13: 978-0-321-83387-7

ISBN-10: 0-321-83387-2

Library of Congress Cataloging-in-Publication Data will be inserted once available.

Printed in the United States of America

First Printing March 2013

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Acquisitions Editor
Mark Taber

Managing Editor
Sandra Schroeder

Senior Project Editor
Tonya Simpson

Copy Editor
Water Crest Publishing

Indexer
Heather McNeill

Proofreader
Jess DeGabriele

Technical Editor
Stephen Frein

Publishing Coordinator
Vanessa Evans

Designer
Chuti Prasertsith

Compositor
Bumpy Design

Contents at a Glance

Note: *Appendixes G, H, and I are located online and are accessible either by registering this book at* ***informit.com/register*** *or by visiting* ***www.kitebird.com/mysql-book****.*

Table of Contents

Part III: MySQL Administration

Note: *Appendixes G, H, and I are located online and are accessible either by registering this book at **informit.com/register** or by visiting **www.kitebird.com/mysql-book**.*

About the Author

Paul DuBois is a writer, database administrator, and leader in the open source and MySQL communities. He has contributed to the online documentation for MySQL and is the author of *MySQL and Perl for the Web* (New Riders), *MySQL Cookbook*, *Using csh and tcsh*, and *Software Portability with imake* (O'Reilly). He is currently a technical writer with the MySQL documentation team at Oracle Corporation.

Acknowledgments

My technical reviewer, Stephen Frein, provided good insights and suggestions for improvement. In addition, because this edition would not have been possible without the previous ones, my continued thanks go to everyone listed in those editions who served as technical reviewer or who patiently answered my questions.

The staff at Pearson responsible for this edition were Mark Taber, acquisitions editor; Tonya Simpson, project editor; Sarah Kearns, copy editor; Kim Scott, compositor; Jess DeGabriele, proofreader; Heather McNeill, indexer; and Chuti Prasertsith, cover designer. My thanks to each of them.

Thanks to my wife Karen for her support and encouragement throughout the production of this edition.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or email address.

Email: feedback@developers-library.info

Mail: Reader Feedback
Addison-Wesley Developer's Library
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at **www.informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

Your purchase of this book includes access to a free online edition for 45 days through the Safari Books Online subscription service. Details are on the last page of this book.

Introduction

A relational database management system (RDBMS) is an essential tool in many environments, from uses in business, research, and educational contexts, to content delivery on the Internet. However, despite the importance of a good database system for managing and accessing information resources, many organizations have found them to be out of reach of their financial resources. Historically, database systems have been an expensive proposition, with vendors charging healthy fees both for software and for support. Also, because database engines often had substantial hardware requirements to run with any reasonable performance, the cost was even greater.

Times have changed, on both the hardware and software sides of the picture. Small desktop systems and servers are inexpensive but powerful, and there is a thriving movement devoted to writing high-performance operating systems for them. These operating systems are available free over the Internet or at the cost of an inexpensive CD. They include several BSD Unix derivatives and several distributions of Linux.

Production of free operating systems has proceeded in concert with—and to a large extent has been made possible by—the development of freely available Open Source tools like the `gcc` GNU C compiler; Apache, the most widely used Web server on the Internet; and well-established general-purpose scripting languages such as Perl, PHP, Python, and Ruby. These all stand in contrast to proprietary solutions that lock you into high-priced products from vendors that don't even provide source code.

Database software has become more accessible, too, and Open Source database systems are freely available. One of the most important is MySQL, a SQL client/server relational database management system originating from Scandinavia. MySQL includes an SQL server, client programs for accessing the server, administrative tools, and a programming interface for writing your own programs.

MySQL's roots begin in 1979, with the UNIREG database tool created by Michael "Monty" Widenius for the Swedish company TcX. In 1994, TcX began searching for an RDBMS with an SQL interface for use in developing Web applications. Commercial servers tested were all found too slow for TcX's large tables, and the freely available mSQL lacked features that TcX required. Consequently, Monty began developing a new server.

In 1995, David Axmark of Detron HB began to push for TcX to release MySQL on the Internet. David also worked on the documentation and on getting MySQL to build with the GNU configuration autotools. MySQL 3.11.1 was unleashed on the world in 1996 in the form of binary distributions for Linux and Solaris. The company MySQL AB was formed to provide distributions of MySQL and to offer commercial services. In 2008, Sun Microsystems acquired MySQL AB, and in 2010, Oracle acquired Sun. Today, MySQL is available in both binary and source form and works on many more platforms.

Initially, MySQL became widely popular because of its speed and simplicity. But there was criticism, too: It lacked features such as transactions and foreign key support. MySQL continued to develop, adding not only those features but others such as replication, subqueries, stored routines, triggers, and views.

These capabilities take MySQL into the realm of enterprise applications. As a result, people who once would have considered only "big iron" database systems for their applications now give serious consideration to MySQL, which runs on anything from modest hardware all the way up to enterprise servers. Its performance rivals any database system you care to put up against it, and it can handle large databases with billions of rows. In the business world, MySQL's presence continues to increase as companies discover it capable of handling their database needs, with the cost for commercial licensing and support a fraction of what they are used to paying.

MySQL lies squarely within the picture that unfolds before us: freely available operating systems running on powerful but inexpensive hardware, putting substantial processing power and capabilities in the hands of businesses and individuals on a wider variety of systems than ever before. This lowering of the economic barriers to computing puts the power of a high-performance RDBMS to work for more organizations than at any time in the past, for very little cost. This is true for individuals as well. For example, I use MySQL with Perl, PHP, and Apache on my Apple laptop running Mac OS X. This enables me to carry my work with me anywhere. Total cost: the cost of the laptop.

Why Choose MySQL?

Several free or low-cost database management systems are available from which to choose, such as MySQL, PostgreSQL, or SQLite. When you compare MySQL with other database systems, think about what's most important to you. Performance, features (such as SQL conformance or extensions), support, licensing conditions, and price all are factors to take into account. Given these considerations, MySQL has many attractive qualities:

- **Speed.** MySQL is fast, and getting faster; see `http://www.mysql.com/why-mysql/benchmarks`. There have been many improvements recently, particularly within InnoDB (which is now the default storage engine) and the query optimizer.
- **Ease of use.** MySQL is a high-performance but relatively simple database system and is much less complex to set up and administer than larger systems.
- **Query language support.** MySQL understands SQL (Structured Query Language), the standard language of choice for all modern database systems.

- **Capability.** The MySQL server is multi-threaded, so many clients can connect to it at the same time. Each client can use multiple databases simultaneously. You can access MySQL interactively using several interfaces that let you enter queries and view the results: command-line clients, Web browsers, or GUI clients. In addition, programming interfaces are available for many languages, such as C, Perl, Java, PHP, Python, and Ruby. You can also access MySQL using applications that support ODBC and .NET (protocols developed by Microsoft). This gives you the choice of using prepackaged client software or writing your own for custom applications.
- **Connectivity and security.** MySQL is fully networked, and databases can be accessed from anywhere on the Internet, so you can share your data with anyone, anywhere. But MySQL has access control so that one person who shouldn't see another's data cannot. To provide additional security, MySQL supports encrypted connections using the Secure Sockets Layer (SSL) protocol.
- **Portability.** MySQL runs on many varieties of Unix and Linux, as well as on other systems such as Windows. MySQL runs on hardware from small devices such as routers and personal computers up to high-end servers with many CPUs and huge amounts of memory.
- **Availability and cost.** MySQL is an Open Source project available under multiple licensing terms. First, it is available under the terms of the GNU General Public License (GPL). This means that MySQL is available without cost for most in-house uses. Second, for organizations that prefer or require formal arrangements or that do not want to be bound by the conditions of the GPL, commercial licenses are available.
- **Open distribution and source code.** MySQL is easy to obtain; just use your Web browser. If you don't understand how something works, are curious about an algorithm, or want to perform a security audit, you can get the source code and examine it. If you think you've found a bug, please report it; the developers want to know.

What about support? Good question; a database system isn't much use if you can't get help for it. This book is one form of assistance, and I like to think that it's useful in that regard. (That the book has reached its fifth edition suggests that it accomplishes that goal.) There are other resources open to you as well, and you'll find that MySQL has good support:

- The MySQL Reference Manual is included in MySQL distributions, and is easily accessible online. The Reference Manual regularly receives good marks in the MySQL user community. This is important; the value of a good product is diminished if no one can figure out how to use it.
- Technical support contracts and educational resources such as training classes are available from Oracle.
- MySQL mailing lists and forums are invaluable support resources that anyone may access. These have many helpful participants, including several MySQL developers.

The MySQL community, developers and nondevelopers alike, is very responsive. Answers to questions on the mailing lists often arrive within minutes. When bugs are reported, the developers generally fix them quickly, and new releases appear regularly.

If you are in the database-selection process, MySQL is an ideal candidate for evaluation. You can try it with no risk or financial commitment. Time for installation and setup is less than for many other systems. If you get stuck, you can use the mailing lists to get help.

Perhaps you're currently running another database system but feel constrained by it: Performance of your current system is a concern; it's proprietary and you don't like being locked into it; you'd like to run on hardware that's not supported by your current system; your software is provided in binary-only format but you want to have the source available; or maybe it just costs too much! All of these are reasons to look into MySQL. Use this book to familiarize yourself with MySQL's capabilities, contact the MySQL sales crew, ask questions on the mailing lists, and you'll find the answers you need to make a decision.

What You Can Expect from This Book

You'll learn how to use MySQL effectively so that you can get your work done more productively. You'll be able to figure out how to get your information into a database, and you'll learn how to get it back out by formulating queries that answer the questions you want to ask of that data.

You need not be a programmer to understand or use SQL. This book shows you how it works. But there's more to understanding how to use a database system properly than knowing SQL syntax. This book emphasizes MySQL's unique capabilities and shows how to use them.

You'll also see how MySQL integrates with other tools. The book shows how to write your own programs that access MySQL databases, and you'll learn to use MySQL with Perl and PHP to generate dynamic Web pages created from the result of database queries.

If you'll be responsible for administering a MySQL installation, this book will tell you what your duties are and how to carry them out. You'll learn how to create user accounts, perform database backups, set up replication, and make sure your site is secure.

Road Map to This Book

This book has four parts. The first concentrates on general concepts of database use. The second focuses on writing your own programs that use MySQL. The third is for readers who have administrative duties. The fourth provides a set of reference appendixes.

Part I: General MySQL Use

- Chapter 1, "Getting Started with MySQL." Discusses how MySQL can be useful to you, provides a tutorial that introduces the interactive `mysql` client program, covers the basics of SQL, and demonstrates MySQL's general capabilities.

- Chapter 2, "Using SQL to Manage Data." Every major RDBMS now available understands SQL, but every database engine implements a slightly different SQL dialect. This chapter discusses SQL with particular emphasis on those features that make MySQL distinctive.
- Chapter 3, "Data Types." Discusses the data types that MySQL provides for storing your information, properties and limitations of each type, when and how to use them, expression evaluation, and type conversion.
- Chapter 4, "Views and Stored Programs." How to write and use SQL objects that are stored on the server side. These include views (virtual tables) and stored programs (functions and procedures, triggers, and events).
- Chapter 5, "Query Optimization." How to make your queries run faster.

Part II: Using MySQL Programming Interfaces

- Chapter 6, "Introduction to MySQL Programming." Discusses some of the application programming interfaces (APIs) for MySQL and provides a general comparison of the APIs that the book covers in detail.
- Chapter 7, "Writing MySQL Programs Using C." How to write C programs using the API provided by the MySQL C client library.
- Chapter 8, "Writing MySQL Programs Using Perl DBI." How to write Perl scripts using the DBI module. Covers standalone command-line scripts and scripts for Web site programming.
- Chapter 9, "Writing MySQL Programs Using PHP." How to use the PHP scripting language and the PHP Data Objects (PDO) database-access extension to write dynamic Web pages that access MySQL databases.

Part III: MySQL Administration

- Chapter 10, "Introduction to MySQL Administration." An overview of the database administrator's duties and what you should know to run a MySQL site successfully.
- Chapter 11, "The MySQL Data Directory." An in-depth look at the organization and contents of the data directory, the area under which MySQL stores databases, logs, and status files.
- Chapter 12, "General MySQL Administration." How to make sure your operating system starts and stops the MySQL server properly when your system comes up and shuts down. Also discusses configuring storage engines, tuning the server, log maintenance, and running multiple servers.
- Chapter 13, "Security and Access Control." What you need to know to make your MySQL installation safe from intrusion, both from other users on the server host and from clients connecting over the network. Discusses how to set up MySQL user accounts, explains the structure of the grant tables that control client access to the MySQL server, and describes how to set up your server to support secure connections over SSL.

- Chapter 14, "Database Maintenance, Backups, and Replication." Discusses how to reduce the likelihood of disaster through preventive maintenance, how to back up your databases, how to perform crash recovery if disaster strikes in spite of your preventive measures, and how to set up replication servers.

Part IV: Appendixes

- Appendix A, "Software Required to Use This Book." Where to get the major tools and sample database files described in the book.
- Appendix B, "Data Type Reference." The characteristics of MySQL's data types.
- Appendix C, "Operator and Function Reference." The operators and functions that are used to write expressions in SQL statements.
- Appendix D, "System, Status, and User Variable Reference." Describes each variable maintained by the MySQL server, and how to use your own variables in SQL statements.
- Appendix E, "SQL Syntax Reference." Describes each SQL statement supported by MySQL.
- Appendix F, "MySQL Program Reference." The programs provided in MySQL distributions.
- Appendix G, "C API Reference." The data types and functions in the MySQL C client library.
- Appendix H, "Perl DBI API Reference." The methods and attributes provided by the Perl DBI module.
- Appendix I, "PHP API Reference." The methods provided for MySQL support in PHP by the PDO extension.

How to Read This Book

Whatever part of the book you happen to be reading, it's best to try the examples as you go along. That means you should do two things:

- If MySQL isn't installed on your system, install it or ask someone to do so for you.
- Get the files needed to set up the `sampdb` sample database used throughout the book.

Appendix A, "Software Required to Use This Book," indicates where to obtain all the necessary components.

If you're new to MySQL or SQL, begin with Chapter 1, "Getting Started with MySQL." It provides you with a tutorial introduction that grounds you in basic MySQL and SQL concepts and brings you up to speed for the rest of the book. Then proceed to Chapter 2, "Using SQL to Manage Data," Chapter 3, "Data Types," and Chapter 4, "Views and Stored Programs," to find out how to describe and manipulate your own data so that you can exploit MySQL's capabilities for your own applications.

If you already know some SQL, you should still read Chapter 2, "Using SQL to Manage Data," and Chapter 3, "Data Types." SQL implementations vary, and you'll want to find out what makes MySQL's implementation distinctive in comparison to others with which you may be familiar. If you have experience with MySQL but need more background on performing particular tasks, use the book as a reference, looking up topics on a need-to-know basis. You'll find the reference appendixes especially useful.

If you're interested in writing your own programs to access MySQL databases, read the API chapters, beginning with Chapter 6, "Introduction to MySQL Programming." To produce a Web-based front end to your databases for easier access to them, or, conversely, to provide a database back end for your Web site to enhance your site with dynamic content, check out Chapter 8, "Writing MySQL Programs Using Perl DBI," and Chapter 9, "Writing MySQL Programs Using PHP."

If your responsibilities include administering a MySQL installation, read the chapters beginning with Chapter 10, "Introduction to MySQL Administration."

If you're evaluating MySQL to find out how it compares to your current RDBMS, several parts of the book are useful. Read the SQL syntax and data type chapters in Part I to compare MySQL to the version of SQL that you're used to, the programming chapters in Part II if you need to write custom applications, and the administrative chapters in Part III to assess the level of administrative support a MySQL installation requires. This information is also useful if you're not currently using a database but are performing a comparative analysis of MySQL along with other database systems for the purpose of choosing one of them.

Versions of Software Covered in This Book

The first edition of this book covered MySQL 3.22 and the beginnings of MySQL 3.23. The second edition expanded that range to include MySQL 4.0 and the first release of MySQL 4.1. The third edition covered MySQL 4.1 and the initial releases of MySQL 5.0. The fourth edition covered MySQL 5.0 and the initial releases of MySQL 5.1.

For this fifth edition, the baseline is MySQL 5.5. That is, the book covers MySQL 5.5 and the early releases of MySQL 5.6. Most of this book still applies if you have a version older than 5.5, but differences specific to older versions usually are not explicitly noted.

The MySQL 5.5 series has reached General Availability (GA) status, which means that it is suitable for production environments. There have been many changes compared to earlier pre-production 5.5 releases, so use the most recent version if possible (5.5.30 as I write). The MySQL 5.6 series currently is a development series (not intended for production use yet) but will reach GA status soon, and may have done so by the time you read this.

For information about older versions, check the MySQL Web site at `http://dev.mysql.com/doc`, where you can access the Reference Manual for each version.

When updating each edition with new material, it's always a challenge to keep the length down. In the interest of space, I have removed some information present in previous editions. The most pervasive change is that InnoDB is now the default storage engine (not MyISAM), so in keeping with the greater emphasis on InnoDB, there is less on MyISAM. Other more minor storage engines such as FEDERATED and BLACKHOLE are mentioned only in passing. I have removed information about `libmysqld` (the embedded server), `mysqlhotcopy`, `myisampack`, spatial data types and functions, and replaced detailed installation material with more general instructions. For information about any of those topics, I recommend the MySQL Reference Manual.

I also draw your attention to some other topics not covered in this book:

- The MySQL Connectors, which provide client access for Java, ODBC, and .NET programs.
- The NDB storage engine and MySQL Cluster, which provide in-memory storage, high availability, and redundancy. See the MySQL Reference Manual for details.
- The graphical user interface (GUI) tool, MySQL Workbench, which helps you use MySQL in a windowing environment.
- MySQL Enterprise, the commercial version of MySQL that includes features such as MySQL Enterprise Monitor that provides server monitoring and diagnostic capabilities, and MySQL Enterprise Backup for hot backups.

To acquire any of these products or see their documentation, visit `http://www.mysql.com/products` or `http://dev.mysql.com/doc`.

For the other major software packages discussed in the book, any recent versions should be sufficient for the examples shown. The following table shows the current versions at the time of writing.

Package	Version
Perl DBI module	1.623
Perl DBD::mysql module	4.020
PHP	5.4.10
Apache	2.4.3
CGI.pm	3.63

All software discussed in this book is available on the Internet. Appendix A, "Software Required to Use This Book," provides assistance for getting MySQL, Perl DBI, PHP and PDO, Apache, and CGI.pm onto your system. The appendix also contains instructions for obtaining the `sampdb` sample database that is used in examples throughout the book and contains the programs developed in the programming chapters.

If you are using Windows, I assume that you have Windows 2000 or newer. Some features covered in this book, such as named pipes and Windows services, are not available in older versions.

Conventions Used in This Book

This book uses the following typographical conventions:

- `Monospaced font` indicates hostnames, filenames, directory names, commands, options, and Web sites.
- **`Bold monospaced font`** is used in command examples to indicate input that you type.
- *`Italic monospaced font`* is used in commands to indicate where you should substitute a value of your own choosing.

Interactive examples assume that you enter commands by typing them into a terminal window or console window. To provide context, the prompt in command examples indicates the program from which you run the command. For example, SQL statements that are issued from within the `mysql` client program are shown preceded by the `mysql>` prompt. For commands that you issue from your command interpreter, the `%` prompt usually is used. In general, this prompt indicates commands that can be run either on Unix or Windows, although the particular prompt you see will depend on your command interpreter. (The command interpreter is your login shell on Unix, or `cmd.exe` on Windows.) More specialized command-line prompts are `#`, which indicates a command run on Unix as the `root` user with `su` or `sudo`, and `C:\>` to indicate a command intended specifically for Windows.

The following example shows a command that should be entered from your command interpreter. The `%` indicates the prompt (which you do not type). To issue the command, you'd enter the boldface characters as shown, and substitute your own username for the italic word:

```
% mysql --user=user_name sampdb
```

In SQL statements, SQL keywords and function names are written in uppercase. Database, table, and column names generally appear in lowercase.

In syntax descriptions, square brackets (`[]`) indicate optional information. In lists of alternatives, vertical bar (`|`) is used as a separator between items. A list enclosed within `[]` is optional and indicates that an item may be chosen from the list. A list enclosed within `{}` is mandatory and indicates that an item must be chosen from the list.

Additional Resources

If you have a question that this book doesn't answer, where should you turn? Useful documentation resources include the Web sites for the software you need help with, shown in the following table.

Package	Primary Web Site
MySQL	`http://dev.mysql.com/doc`
Perl DBI	`http://dbi.perl.org`
PHP	`http://www.php.net`
Apache	`http://httpd.apache.org`
CGI.pm	`http://search.cpan.org/dist/CGI.pm`

Those sites provide information such as reference manuals, frequently asked-question (FAQ) lists, and mailing lists:

- **Reference manuals.** The primary documentation included with MySQL itself is the Reference Manual. It's available in several formats, including online and downloadable versions.
- **Manual pages.** Documentation for the DBI module and its MySQL-specific driver, DBD::mysql, can be read from the command line with the `perldoc` command. Try `perldoc DBI` and `perldoc DBD::mysql`. The DBI document provides general concepts, and the MySQL driver document discusses capabilities specific to MySQL.
- **FAQs.** There are frequently-asked-question lists for DBI, PHP, and Apache.
- **Mailing lists.** Several mailing lists centering around the software discussed in this book are available. It's a good idea to subscribe to the ones that deal with the tools you want to use. It's also a good idea to use the archives for those lists that have them. When you're new to a tool, you will have many of the same questions that have been asked (and answered) many times, and there is no reason to ask again when you can find the answer with a quick search of the archives.

  Instructions for subscribing to the mailing lists vary. The following table indicates where you can find the necessary information.

Package	Mailing List Instructions
MySQL	`http://lists.mysql.com`
Perl DBI	`http://dbi.perl.org/support`
PHP	`http://www.php.net/mailing-lists.php`
Apache	`http://httpd.apache.org/lists.html`

- **Ancillary Web sites.** Besides the official Web sites, some of the tools discussed here have ancillary sites that provide more information, such as sample source code or topical articles. Check for a "Links" area on the official site you're visiting.

2

Using SQL to Manage Data

The MySQL server understands Structured Query Language (SQL). Therefore, SQL is the means by which you tell the server how to perform data management operations, and fluency with it is necessary for effective communication. When you use a program such as the `mysql` client, it functions primarily as a way for you to send SQL statements to the server to be executed. If you write programs in a language that has a MySQL interface, such as the Perl DBI module or PHP PDO extension, these interfaces enable you to communicate with the server by issuing SQL statements.

Chapter 1, "Getting Started with MySQL," presented a tutorial introducing many of MySQL's capabilities, including some basic use of SQL. We'll build on that material here to go into more detail on several topics:

- Changing the SQL mode to affect server behavior
- Referring to elements of databases
- Using multiple character sets
- Creating and destroying databases, tables, and indexes
- Obtaining information about databases and their contents
- Retrieving data using joins, subqueries, and unions
- Using multiple-table deletes and updates
- Performing transactions that enable statements to be grouped or canceled
- Setting up foreign key relationships
- Using the `FULLTEXT` search engine

The items just listed cover a broad range of topics of what you can do with SQL. Other chapters provide additional SQL-related information:

- Chapter 4, "Views and Stored Programs," discusses how to create and use views (virtual tables that provide alternative ways of looking at data) and stored programs (functions and procedures, triggers, and events).

- Chapter 12, “General MySQL Administration,” describes how to use administrative statements such as `GRANT` and `REVOKE` to manage user accounts. It also discusses the privilege system that controls what operations accounts are permitted to perform.
- Appendix E, “SQL Syntax Reference,” shows the syntax for SQL statements implemented by MySQL and the privileges required to use them. It also covers the syntax for using comments in your SQL statements.

See also the MySQL Reference Manual, especially for changes made in recent versions of MySQL.

2.1 The Server SQL Mode

The MySQL SQL mode affects several aspects of SQL statement execution, and the server has a system variable named `sql_mode` that enables you to configure this mode. The variable can be set globally to affect all clients, and each individual client can change the mode to affect its own session with (connection to) the server. This means that any client can change how the server treats it without impact on other clients.

The SQL mode affects behaviors such as handling of invalid values during data entry and identifier quoting. The following list describes a few of the possible mode values:

- `STRICT_ALL_TABLES` and `STRICT_TRANS_TABLES` enable “strict” mode. In strict mode, the server is more restrictive about accepting bad data values. (Specifically, it rejects bad values rather than changing them to the closest legal value.)
- `TRADITIONAL` is a composite mode. It is like strict mode, but enables other modes that impose additional constraints for even stricter data checking. Traditional mode causes the server to behave like more traditional SQL servers with regard to how it handles bad data values.
- `ANSI_QUOTES` tells the server to recognize double quote as an identifier quoting character.
- `PIPES_AS_CONCAT` causes `||` to be treated as the standard SQL string concatenation operator rather than as a synonym for the `OR` operator.
- `ANSI` is another composite mode. It turns on `ANSI_QUOTES`, `PIPES_AS_CONCAT`, and several other mode values that cause the server to conform more closely to standard SQL.

When you set the SQL mode, specify a value consisting of one or more mode values separated by commas, or an empty string to clear the value. Mode values are not case sensitive.

To set the SQL mode when you start the server, set the `sql_mode` system variable on the `mysqld` command line or in an option file. On the command line, you might use a setting like one of these:

```
--sql_mode="TRADITIONAL"
--sql_mode="ANSI_QUOTES,PIPES_AS_CONCAT"
```

To change the SQL mode at runtime, set the `sql_mode` system variable with a `SET` statement. Any client can set its own session-specific SQL mode:

```
SET sql_mode = 'TRADITIONAL';
```

To set the SQL mode globally, add the `GLOBAL` keyword:

```
SET GLOBAL sql_mode = 'TRADITIONAL';
```

Setting the global variable requires the `SUPER` administrative privilege. The global value becomes the default SQL mode for clients that connect afterward.

To determine the current value of the session or global SQL mode, use these statements:

```
SELECT @@SESSION.sql_mode;
SELECT @@GLOBAL.sql_mode;
```

The value returned consists of a comma-separated list of enabled modes, or an empty value if no modes are enabled.

Section 3.3, "How MySQL Handles Invalid Data Values," discusses the SQL mode values that affect handling of erroneous or missing values during data entry. Appendix D, "System, Status, and User Variable Reference," describes the full set of permitted mode values for the `sql_mode` variable. For additional information about using system variables, see Section 12.3.1, "Checking and Setting System Variable Values."

2.2 MySQL Identifier Syntax and Naming Rules

Almost every SQL statement uses identifiers in some way to refer to a database or its constituent elements such as tables, views, columns, indexes, stored routines, triggers, or events. When you refer to elements of databases, identifiers must conform to the following rules.

Legal characters in identifiers. Unquoted identifiers may consist of latin letters `a-z` in any lettercase, digits `0-9`, dollar, underscore, and Unicode extended characters in the range `U+0080` to `U+FFFF`. Identifiers can start with any character that is legal in an identifier, including a digit. However, an unquoted identifier cannot consist entirely of digits because that would make it indistinguishable from a number. MySQL's support for identifiers that begin with a number is somewhat unusual among database systems. If you use such an identifier, take particular care if it contains an '`E`' or '`e`' because those characters can lead to ambiguous expressions. For example, the expression `23e + 14` (with spaces surrounding the '`+`' sign) means column `23e` plus the number `14`, but what about `23e+14`? Does it mean the same thing, or is it a number in scientific notation?

Identifiers can be quoted (delimited) within backtick characters ('`` ` ``'), which permits use of any character except a NUL byte or Unicode supplementary characters (`U+10000` and up):

```
CREATE TABLE `my table` (`my-int-column` INT);
```

Quoting is useful when an identifier is an SQL reserved word or contains spaces or other special characters. Quoting an identifier also enables it to be entirely numeric, something not true of unquoted identifiers. To include an identifier quote character within a quoted identifier, double it.

Your operating system might impose additional constraints on database and table identifiers. See Section 11.2.6, "Operating System Constraints on Database Object Names."

Aliases for column and table names can be fairly arbitrary. You should quote an alias within identifier quoting characters if it is an SQL reserved word, is entirely numeric, or contains spaces or other special characters. Column aliases also can be quoted with single quotes or double quotes.

Server SQL mode. If the `ANSI_QUOTES` SQL mode is enabled, you can quote identifiers with double quotes (although backticks still are permitted).

```
CREATE TABLE "my table" ("my-int-column" INT);
```

Enabling `ANSI_QUOTES` has the additional effect that string literals *must* be written using single quotes. If you use double quotes, the server interprets the value as an identifier, not as a string.

Names of built-in functions normally are not reserved and can be used as identifiers without quotes. However, if the `IGNORE_SPACE` SQL mode is enabled, function names become reserved and must be quoted if used as identifiers.

For instructions on setting the SQL mode, see Section 2.1, "The Server SQL Mode."

Identifier length. Most identifiers have a maximum length of 64 characters. The maximum length for aliases is 256 characters.

Identifier qualifiers. Depending on context, an identifier might need to be qualified to clarify what it refers to. To refer to a database, just specify its name:

```
USE db_name;
SHOW TABLES FROM db_name;
```

To refer to a table, you have two choices:

- A fully qualified table name consists of a database identifier and a table identifier:

  ```
  SHOW COLUMNS FROM db_name.tbl_name;
  SELECT * FROM db_name.tbl_name;
  ```

- A table identifier by itself refers to a table in the default (current) database. If `sampdb` is the default database, the following statements are equivalent:

  ```
  SELECT * FROM member;
  SELECT * FROM sampdb.member;
  ```

If no database is selected, it is an error to refer to a table without a database qualifier because the database to which the table belongs is unknown.

The same considerations about qualifying table names apply to names of views (which are "virtual" tables) and stored programs.

To refer to a table column, you have three choices:

- A name written as *db_name.tbl_name.col_name* is fully qualified.
- A partially qualified name written as *tbl_name.col_name* refers to a column in the named table in the default database.
- An unqualified name written simply as *col_name* refers to whatever table the surrounding context indicates. The following two queries use the same column names, but the context supplied by the FROM clause of each statement indicates the table from which to select the columns:

```
SELECT last_name, first_name FROM president;
SELECT last_name, first_name FROM member;
```

Usually, it's unnecessary to supply fully qualified names, although it's always legal to do so. If you select a database with a USE statement, it becomes the default database for subsequent statements and is implicit in every unqualified table reference. If you write a SELECT statement that refers to only one table, that table is implicit for every column reference in the statement. It's necessary to qualify identifiers only when a table or database cannot be determined from context. For example, if a statement refers to tables from multiple databases, you must reference any table not in the default database using *db_name.tbl_name* syntax to let MySQL know which database contains the table. Similarly, if a query uses multiple tables and refers to a column name that is used in more than one table, qualify the column identifier with a table identifier to make it clear which column you mean.

If you use quotes when referring to a qualified name, quote individual identifiers within the name separately. For example:

```
SELECT * FROM `sampdb`.`member` WHERE `sampdb`.`member`.`member_id` > 100;
```

Do not quote the name as a whole. This statement is incorrect:

```
SELECT * FROM `sampdb.member` WHERE `sampdb.member.member_id` > 100;
```

The requirement that a reserved word be quoted if used as an identifier is waived if the word follows a qualifier period because context then dictates that the reserved word is an identifier.

2.3 Case Sensitivity in SQL Statements

Case sensitivity rules in SQL statements vary for different statement elements, and also depend on what you are referring to and the operating system of the machine on which the server is running.

SQL keywords and function names. Keywords and function names are not case sensitive. They can be given in any lettercase. The following statements are equivalent:

```
SELECT NOW();
select now();
sElEcT nOw();
```

Database, table, and view names. MySQL represents databases and tables using directories and files in the underlying filesystem on the server host. As a result, the default case sensitivity of database and table names depends on how the operating system on that host treats filenames. Windows filenames are not case sensitive, so a MySQL server running on Windows does not treat database and table names as case sensitive. Servers running on Unix usually treat database and table names as case sensitive because Unix filenames are case sensitive. An exception is that names in Mac OS X Extended filesystems can be case insensitive.

MySQL represents each view using a file, so the preceding remarks about tables also apply to views.

Stored program names. Stored function and procedure names and event names are not case sensitive. Trigger names are case sensitive, which differs from standard SQL.

Column and index names. Column and index names are not case sensitive in MySQL. The following statements are equivalent:

```
SELECT name FROM student;
SELECT NAME FROM student;
SELECT nAmE FROM student;
```

Alias names. By default, table aliases are case sensitive. You can specify an alias in any lettercase (upper, lower, or mixed), but if you use it multiple times in a statement, you must use the same lettercase each time. If the `lower_case_table_names` system variable is nonzero, table aliases are not case sensitive.

String values. Case sensitivity of a string value depends on whether it is a binary or nonbinary string, and, for a nonbinary string, on the collation of its character set. This is true for literal strings and the contents of string columns. For further information, see Section 3.1.2, “String Values.”

You should consider lettercase issues when you create databases and tables on a machine with case sensitive filenames if you might someday move them to a machine where filenames are not case sensitive. Suppose that you create two tables named `abc` and `ABC` on a Unix server where those names are considered distinct. You would have problems moving the tables to a Windows machine: `abc` and `ABC` are not distinguishable because names are not case sensitive. You would also have trouble replicating the tables from a Unix master server to a Windows slave server.

To avoid having case sensitivity become an issue, pick a given lettercase and always create databases and tables using names in that lettercase. Then case of names won’t be a problem if you move a database to a different server. I recommend lowercase, particularly if you are using InnoDB tables, because InnoDB stores database and table names internally in lowercase.

To force creation of databases and tables with lowercase names even if not specified that way in `CREATE` statements, configure the server by setting the `lower_case_table_names` system variable. For more information, see Section 11.2.6, “Operating System Constraints on Database Object Names.”

Regardless of whether a database or table name is case sensitive on your system, you must refer to it using the same lettercase throughout a given query. That is not true for SQL keywords, function names, or column and index names, all of which may be referred to in varying lettercase style throughout a query.

2.4 Character Set Support

MySQL supports multiple character sets, and character sets can be specified independently at the server, database, table, column, or string constant level. For example, if you want a table's columns to use `latin1` by default, but also to include a Hebrew column and a Greek column, you can do that. In addition, you can explicitly specify collations (sorting orders). It is possible to find out what character sets and collations are available, and to convert data from one character set to another.

This section provides general background on using character set support in MySQL. Chapter 3, "Data Types," provides more specific discussion of character sets, collations, binary versus nonbinary strings, and how to define and work with character-based table columns.

MySQL provides the following character set features:

- The server supports simultaneous use of multiple character sets.
- A given character set can have one or more collations. You can choose the collation most appropriate for your applications.
- Unicode support is provided by the `utf8` and `ucs2` character sets, which include Basic Multilingual Plane (BMP) characters, and the `utf16`, `utf32`, and `utf8mb4` character sets, which include BMP and supplementary characters. MySQL 5.6.1 adds `utf16le`, which is like `utf16` but uses little-endian rather than big-endian encoding.
- You can specify character sets at the server, database, table, column, and string constant level:
  - The server has a default character set.
  - `CREATE DATABASE` enables you to assign the database character set, and `ALTER DATABASE` enables you to change it.
  - `CREATE TABLE` and `ALTER TABLE` have clauses for table- and column-level character set assignment.
  - The character set for string constants is determined by context or can be specified explicitly.
- Several functions and operators are available for converting individual values from one character set to another, and the `CHARSET()` function returns the character set of a value. Similarly, the `COLLATE` operator can be used to alter the collation of a string and the `COLLATION()` function returns the collation of a string.

- `SHOW` statements and `INFORMATION_SCHEMA` tables provide information about the available character sets and collations.
- The server automatically reorders indexes when you change the collation of an indexed character column.

You cannot mix character sets within a string, or use different character sets for different rows of a given column. However, you can implement multi-lingual support by using a Unicode character set (which represents characters for many languages within a single encoding).

2.4.1 Specifying Character Sets

Character set and collation assignments can be made at several levels, from the default used by the server to the character set used for individual strings.

The server's default character set and collation are built in at compile time. You can override them at server startup or at runtime by setting the `character_set_server` and `collation_server` system variables, as described in Section 12.6.2, "Selecting the Default Character Set and Collation." If you specify only the character set, its default collation becomes the server's default collation. If you specify a collation, it must be compatible with the character set. A collation is compatible with a character set if its name begins with the character set name. For example, `utf8_danish_ci` is compatible with `utf8` but not with `latin1`.

In SQL statements that create databases and tables, two clauses specify database, table, and column character set and collation values:

```
CHARACTER SET charset
COLLATE collation
```

`CHARSET` can be used as a synonym for `CHARACTER SET`. *`charset`* is the name of a character set supported by the server, and *`collation`* is the name of one of that character set's collations. These clauses can be specified together or separately. If both are given, the collation name must be compatible with the character set. If only `CHARACTER SET` is given, its default collation is used. If only `COLLATE` is given, the character set is implicit in the first part of the character set name. These rules apply at several levels:

- To specify a default character set and collation for a database when you create it, use this statement:

  ```
  CREATE DATABASE db_name CHARACTER SET charset COLLATE collation;
  ```

  If no character set or collation is given, the database uses the server defaults.
- To specify a default character set and collation for a table, use `CHARACTER SET` and `COLLATE` table options at table creation time:

  ```
  CREATE TABLE tbl_name (...) CHARACTER SET charset COLLATE collation;
  ```

  If no character set or collation is given, the table uses the database defaults.

- Columns in a table can be assigned a character set and collation explicitly with `CHARACTER SET` and `COLLATE` attributes. For example:

  ```
  c CHAR(10) CHARACTER SET charset COLLATE collation
  ```

  If no character set or collation is given, the column uses the table defaults. These attributes apply to the `CHAR`, `VARCHAR`, `TEXT`, `ENUM`, and `SET` data types.

It's also possible to sort string values according to a specific collation by using the `COLLATE` operator. For example, if `c` is a `latin1` column that has a collation of `latin1_swedish_ci`, but you want to order it using Spanish sorting rules, do this:

```
SELECT c FROM t ORDER BY c COLLATE latin1_spanish_ci;
```

2.4.2 Determining Character Set Availability and Current Settings

To find out which character sets and collations are available, use these statements:

```
SHOW CHARACTER SET;
SHOW COLLATION;
```

Each statement supports a `LIKE` clause that narrows the results to those character set or collation names matching a pattern. For example, the following statements list the Latin-based character sets and the collations available for the `utf8` character set:

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | cp1252 West European        | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
mysql> SHOW COLLATION LIKE 'utf8%';
+-----------------------+---------+-----+---------+----------+---------+
| Collation             | Charset | Id  | Default | Compiled | Sortlen |
+-----------------------+---------+-----+---------+----------+---------+
| utf8_general_ci       | utf8    |  33 | Yes     | Yes      |       1 |
| utf8_bin              | utf8    |  83 |         | Yes      |       1 |
| utf8_unicode_ci       | utf8    | 192 |         | Yes      |       8 |
| utf8_icelandic_ci     | utf8    | 193 |         | Yes      |       8 |
| utf8_latvian_ci       | utf8    | 194 |         | Yes      |       8 |
| utf8_romanian_ci      | utf8    | 195 |         | Yes      |       8 |
| utf8_slovenian_ci     | utf8    | 196 |         | Yes      |       8 |
...
```

Collation names always begin with the character set name. Each character set has at least one collation, and one of them is its default collation.

Information about the available character sets or collations can also be obtained from the CHARACTER_SETS or COLLATIONS table in the INFORMATION_SCHEMA database (see Section 2.7, "Obtaining Database Metadata").

To display the server's current character set and collation settings, use SHOW VARIABLES:

```
mysql> SHOW VARIABLES LIKE 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | utf8   |
| character_set_connection | utf8   |
| character_set_database   | latin1 |
| character_set_filesystem | binary |
| character_set_results    | utf8   |
| character_set_server     | latin1 |
| character_set_system     | utf8   |
+--------------------------+--------+
mysql> SHOW VARIABLES LIKE 'collation\_%';
+----------------------+-------------------+
| Variable_name        | Value             |
+----------------------+-------------------+
| collation_connection | utf8_general_ci   |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+----------------------+-------------------+
```

Several of these system variables affect how a client communicates with the server after establishing a connection. For details, refer to Section 3.1.2.2, "Character Set-Related System Variables."

2.4.3 Unicode Support

One of the reasons there are so many character sets is that different character encodings have been developed for different languages. This presents several problems. For example, a given character that is common to several languages might be represented by different numeric values in different encodings. Also, different languages require different numbers of bytes to represent characters. The latin1 character set is small enough that every character fits in a single byte, but languages such as those used in Japan and China contain so many characters that they require multiple bytes per character.

Unicode deals with these issues by providing a unified character-encoding system within which character sets for all languages can be represented in a consistent manner.

The utf8 and ucs2 Unicode character sets include only characters in the Basic Multilingual Plane (BMP), which is limited to 65,536 characters. They do not support supplementary characters outside the BMP.

- The ucs2 character set corresponds to the Unicode UCS-2 encoding. It represents each character using 2 bytes, most significant byte first. UCS is an abbreviation for Universal Character Set.
- The utf8 character set has a variable-length format that represents characters using from 1 to 3 bytes. It corresponds to the Unicode UTF-8 encoding. UTF is an abbreviation for Unicode Transformation Format.

Beginning with MySQL 5.5.3, other Unicode character sets are available that include supplementary characters in addition to BMP characters.

- The utf16 and utf32 character sets are like ucs2 but with supplementary characters added. For utf16, BMP characters take 2 bytes (as for ucs2) and supplementary characters take 4 bytes. For utf32, all characters take 4 bytes.
- The utf8mb4 character set contains all the utf8 characters (which take 1 to 3 bytes each), but also supplementary characters that take 4 bytes each.

MySQL 5.6.1 adds utf16le, which is like utf16 but uses little-endian rather than big-endian encoding.

2.5 Selecting, Creating, Dropping, and Altering Databases

MySQL provides several database-level statements: USE for selecting a default database, CREATE DATABASE for creating databases, DROP DATABASE for removing them, and ALTER DATABASE for modifying global database characteristics.

The keyword SCHEMA is a synonym for DATABASE in any statement where the latter occurs.

2.5.1 Selecting Databases

The USE statement selects a database to make it the default (current) database for a given session with the server:

```
USE db_name;
```

You must have some access privilege for the database or an error occurs.

It is not strictly necessary to select a database explicitly. You can refer to tables in a database without selecting it first by using qualified names that identify both the database and the table. For example, to retrieve the contents of the president table in the sampdb database without making it the default database, write the query like this:

```
SELECT * FROM sampdb.president;
```

Selecting a database doesn't mean that it must be the default for the duration of the session. You can issue USE statements as necessary to switch between databases. Nor does selecting a database limit you to using tables only from that database. While one database is the default, you can refer to tables in other databases by qualifying their names with the appropriate database identifier.

When you disconnect from the server, any notion by the server of which database was the default for the session disappears. If you connect to the server again, it doesn't remember what database you had selected previously.

2.5.2 Creating Databases

To create a database, use a `CREATE DATABASE` statement:

```
CREATE DATABASE db_name;
```

The database must not already exist, and you must have the `CREATE` privilege for it.

`CREATE DATABASE` supports several optional clauses. The full syntax is as follows:

```
CREATE DATABASE [IF NOT EXISTS] db_name
  [CHARACTER SET charset] [COLLATE collation];
```

By default, an error occurs if you try to create a database that already exists. To suppress this error and create a database only if it does not already exist, add an `IF NOT EXISTS` clause:

```
CREATE DATABASE IF NOT EXISTS db_name;
```

By default, the server character set and collation become the database default character set and collation. To set these database attributes explicitly, use the `CHARACTER SET` and `COLLATE` clauses. For example:

```
CREATE DATABASE mydb CHARACTER SET utf8 COLLATE utf8_icelandic_ci;
```

If `CHARACTER SET` is given without `COLLATE`, the character set default collation is used. If `COLLATE` is given without `CHARACTER SET`, the first part of the collation name determines the character set.

The character set must be one of those supported by the server, such as `latin1` or `sjis`. The collation should be a legal collation for the character set. For further discussion of character sets and collations, see Section 2.4, "Character Set Support."

When you create a database, the MySQL server creates a directory under its data directory that has the same name as the database. The new directory is called the database directory. The server also creates a `db.opt` file in the database directory for storing attributes such as the database character set and collation. When you create a table in the database later, the database defaults become the table defaults if the table definition does not specify its own default character set and collation.

To see the definition for an existing database, use a `SHOW CREATE DATABASE` statement:

```
mysql> SHOW CREATE DATABASE mydb\G
*************************** 1. row ***************************
       Database: mydb
Create Database: CREATE DATABASE `mydb`
                 /*!40100 DEFAULT CHARACTER SET utf8
                 COLLATE utf8_icelandic_ci */
```

2.5.3 Dropping Databases

Dropping a database is as easy as creating one, assuming that you have the `DROP` privilege for it:

```
DROP DATABASE db_name;
```

The `DROP DATABASE` statement is not something to use with wild abandon. It removes the database and all its contents (tables, stored routines, and so forth), which are therefore gone forever unless you have been making regular backups.

A database is represented by a directory under the data directory, and the directory is intended for storage of objects such as tables, views, and triggers. If a `DROP DATABASE` statement fails, the reason most likely is that the database directory contains files not associated with database objects. `DROP DATABASE` will not delete such files, and as a result will not delete the directory, either. This means that the database directory continues to exist and will show up if you issue a `SHOW DATABASES` statement. To really drop the database if this occurs, manually remove any extraneous files and subdirectories from the database directory, then issue the `DROP DATABASE` statement again.

2.5.4 Altering Databases

The `ALTER DATABASE` statement changes a database's global attributes, if you have the `ALTER` privilege for it. Currently, the only such attributes are the default character set and collation:

```
ALTER DATABASE [db_name] [CHARACTER SET charset] [COLLATE collation];
```

The earlier discussion for `CREATE DATABASE` describes the effect of the `CHARACTER SET` and `COLLATE` clauses, at least one of which must be given.

If you omit the database name, `ALTER DATABASE` applies to the default database.

2.6 Creating, Dropping, Indexing, and Altering Tables

MySQL enables you to create tables, drop (remove) them, and change their structure with the `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE` statements. The `CREATE INDEX` and `DROP INDEX` statements enable you to add or remove indexes on existing tables. The following sections provide the details for these statements, but first it's necessary to discuss the storage engines that MySQL supports for managing different types of tables.

2.6.1 Storage Engine Characteristics

MySQL supports multiple storage engines (or "table handlers" as they used to be known). Each storage engine implements tables that have a specific set of properties or characteristics. Table 2.1 briefly describes these storage engines, and later discussion provides more detail about some of them (primarily InnoDB and MyISAM). Others are either less commonly used or, in the case of NDB, require extensive discussion beyond what can be given here. Consequently, the remainder of this book says little about them.

Table 2.1 **MySQL Storage Engines**

Storage Engine	Description
ARCHIVE	Archival storage (no modification of rows after insertion)
BLACKHOLE	Engine that discards writes and returns empty reads
CSV	Storage in comma-separated values format
FEDERATED	Engine for accessing remote tables
InnoDB	Transactional engine with foreign keys
MEMORY	In-memory tables
MERGE	Manages collections of MyISAM tables
MyISAM	The main nontransactional storage engine
NDB	The engine for MySQL Cluster

Some of the engine names have synonyms. MRG_MyISAM and NDBCLUSTER are synonyms for MERGE and NDB, respectively. The MEMORY and InnoDB storage engines originally were known as HEAP and Innobase, respectively. The latter names are still recognized but deprecated.

Originally, the MySQL server was built such that all storage engines to be made available were compiled in. Now the server uses a "pluggable" architecture that enables plugins to be loaded selectively, and many storage engines are built as plugins. This permits the DBA to treat those engines as optional and load only those needed. The plugin interface also permits storage engines from third-party developers to be integrated into the server. For information about this interface, see Section 12.4, "The Plugin Interface."

2.6.1.1 Checking Which Storage Engines Are Available

The engines actually available for a given server depend on your version of MySQL, how the server was configured at build time, and the startup options you use. For information about selecting storage engines, see Section 12.5, "Storage Engine Configuration."

To see which storage engines the server knows about, use the `SHOW ENGINES` statement:

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
...
*************************** 8. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
...
```

The `Support` column value is `YES` or `NO` to indicate that the engine is or is not available, `DISABLED` if the engine is present but turned off, or `DEFAULT` for the storage engine that the server uses by default. The engine designated as `DEFAULT` should be considered available. The `Transactions` column indicates whether an engine supports transactions. `XA` and `Savepoints` indicate whether an engine supports distributed transactions (not covered in this book) and partial transaction rollback.

The `ENGINES` table in the `INFORMATION_SCHEMA` database provides the same information as `SHOW ENGINES`, but since you access it with `SELECT`, you can apply query conditions to select only the information in which you're interested. For example, this query uses the `ENGINES` table to check for available engines that support transactions:

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.ENGINES
    -> WHERE TRANSACTIONS = 'YES';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+
```

2.6.1.2 Table Representation on Disk

Each time you create a table, MySQL creates a disk file that contains the table's format (that is, its definition). The format file has a basename that is the same as the table name and an `.frm` extension. For a table named `t`, the format file is named `t.frm`. The server creates the file in the database directory for the database that the table belongs to. The `.frm` file is an invariant because there is one for every table, no matter which storage engine manages the table. The name of a table as used in SQL statements might differ from the table-name part of the associated `.frm` file if the name contains characters that are problematic in filenames. See

Section 11.2.6, "Operating System Constraints on Database Object Names," for a description of the rules for mapping from SQL names to filenames.

Individual storage engines may also create other files that are unique to the table, to be used for storing the table's content. For a given table, any files specific to it are located in the database directory for the database that contains the table. Table 2.2 shows the filename extensions for table-specific files created by certain storage engines.

Table 2.2 **Table Files Created by Storage Engines**

Storage Engine	Files on Disk
InnoDB	`.ibd` (data and indexes)
MyISAM	`.MYD` (data), `.MYI` (indexes)
CSV	`.CSV` (data), `.CSM` (metadata)

For some storage engines, the format file is the only file specifically associated with a particular table. Other engines may store table content elsewhere than on disk, or may use one or more tablespaces (storage areas shared by multiple tables):

- The MEMORY storage engine stores table contents in memory, not on disk.
- By default, InnoDB stores table data and indexes in its system tablespace. That is, all InnoDB table contents are managed within a shared storage area, not within files specific to a particular table. Alternatively, InnoDB creates `.ibd` files if you configure it to use individual per-table tablespaces.

The following sections characterize the features and behavior of selected MySQL storage engines. For additional information about how engines represent tables physically, see Section 11.2.3, "Representation of Tables in the Filesystem."

2.6.1.3 The InnoDB Storage Engine

The InnoDB storage engine is the default engine in MySQL, unless you have configured your server otherwise. The following list describes some of its features:

- Transaction-safe tables with commit and rollback. Savepoints can be created to enable partial rollback.
- Automatic recovery after a crash.
- Foreign key and referential integrity support, including cascaded delete and update.
- Row-level locking and multi-versioning for good concurrency performance under query mix conditions that include both retrievals and updates.
- As of MySQL 5.6, InnoDB supports full-text searches and `FULLTEXT` indexes.

By default, InnoDB manages tables within a single system tablespace, rather than by using table-specific files like most other storage engines. The tablespace consists of one or more files and can include raw partitions. The InnoDB storage engine, in effect, treats the tablespace as a virtual filesystem within which it manages the contents of all InnoDB tables. Tables thus can exceed the size permitted by the filesystem for individual files. You can also configure InnoDB to use a separate tablespace file for each table. In this case, each table has an `.ibd` file in its database directory.

To configure individual tablespaces, enable the `innodb_file_per_table` system variable, either at server startup or at runtime. Enabling this variable also enables other InnoDB features, such as fast table truncation and row storage formats that offer more efficient table processing for some kinds of data. For more information, see Section 12.5.3.1.4, “Using Individual (Per-Table) InnoDB Tablespaces.”

2.6.1.4 The MyISAM Storage Engine

The MyISAM storage engine offers these features:

- Key compression when storing runs of successive similar string index values. MyISAM also can compress runs of similar numeric index values because numeric values are stored with the high byte first. (Index values tend to vary faster in the low-order bytes, so high-order bytes are more subject to compression.) To enable numeric compression, use the `PACK_KEYS=1` option when creating a MyISAM table.
- More features for `AUTO_INCREMENT` columns than provided by other storage engines. For more information, see Section 3.4, “Working with Sequences.”
- Each MyISAM table has a flag that is set when a table-check operation is performed. MyISAM tables also have a flag indicating whether a table was closed properly when last used. If the server shuts down abnormally or the machine crashes, the flags can be used to detect tables that need to be checked. To do this automatically, start the server with the `myisam_recover_options` system variable set to a value that includes the `FORCE` option. This causes the server to check the table flags whenever it opens a MyISAM table and perform a table repair if necessary. See Section 14.3.1, “Using the Server’s Auto-Recovery Capabilities.”
- Full-text searches and `FULLTEXT` indexes.
- Spatial data types and `SPATIAL` indexes.

2.6.1.5 The MEMORY Storage Engine

The MEMORY storage engine uses tables that are stored in memory and that have fixed-length rows, two properties that make them very fast.

MEMORY tables are temporary in the sense that their contents disappear when the server terminates. That is, a MEMORY table still exists when the server restarts, but will be empty. However, in contrast to temporary tables created with `CREATE TEMPORARY TABLE`, MEMORY tables are visible to other clients.

MEMORY tables have characteristics that enable them to be handled more simply, and thus more quickly:

- By default, MEMORY tables use hashed indexes, which are very fast for equality comparisons but slow for range comparisons. Consequently, hashed indexes are used only for comparisons performed with the `=` and `<=>` equality operators, but not for comparison operators such as `<` or `>`. Hashed indexes also are not used in `ORDER BY` clauses for this reason.
- Rows are stored in MEMORY tables using fixed-length format for easier processing. A consequence is that you cannot use the `BLOB` and `TEXT` variable-length data types. `VARCHAR` is a variable-length type, but is permitted because it is treated internally as `CHAR`, a fixed-length type.

To use a MEMORY table for comparisons that look for a range of values using operators such as `<`, `>`, or `BETWEEN`, you can use `BTREE` indexes instead of hashed indexes. See Section 2.6.4.2, "Creating Indexes," and Section 5.1.3, "Choosing Indexes."

2.6.1.6 The NDB Storage Engine

NDB is MySQL's cluster storage engine. For this storage engine, the MySQL server actually acts as a client to a cluster of other processes that provide access to the NDB tables. Cluster node processes communicate with each other to manage tables in memory. The tables are replicated among cluster processes for redundancy. Memory storage provides high performance, and the cluster provides high availability because it survives failure of any given node.

NDB configuration and use is beyond the scope of this book and is not covered further here. See the MySQL Reference Manual for details.

2.6.1.7 Other Storage Engines

MySQL has several other storage engines that I group here under the "miscellaneous" category:

- The ARCHIVE engine provides archival storage. It's intended for storage of large numbers of rows that are written once and never modified thereafter. For this reason, it supports only a limited number of statements. `INSERT` and `SELECT` work, but `REPLACE` always acts like `INSERT`, and you cannot use `DELETE` or `UPDATE`. Rows are compressed during storage and decompressed during retrieval to save space. An ARCHIVE table can include an indexed `AUTO_INCREMENT` column; other columns cannot be indexed.
- The BLACKHOLE engine creates tables for which writes are ignored and reads return nothing. It is the database equivalent of the Unix `/dev/null` device.
- The CSV engine stores data in comma-separated values format. For each table, it creates a `.CSV` file in the database directory. This is a plain text file in which each table row appears as a single line. The CSV engine does not support indexing.

- The FEDERATED engine provides access to tables that are managed by other MySQL servers. In other words, the contents of a FEDERATED table really are located remotely. For a FEDERATED table, you specify the host where the other server is running and provide the username and password of an account on that server. When you access the FEDERATED table, the local server connects to the remote server using this account.
- The MERGE engine provides a means of grouping a set of MyISAM tables into a single logical unit. Querying a MERGE table in effect queries all the constituent tables. One advantage of this is that you can exceed the maximum table size permitted by the filesystem for individual MyISAM tables. Partitioned tables provide an alternative to MERGE tables and are not limited to MyISAM tables. See Section 2.6.2.5, "Using Partitioned Tables."

2.6.2 Creating Tables

To create a table, use a `CREATE TABLE` statement. You must have the `CREATE` privilege for the table. The full syntax for this statement is complex because there are so many optional clauses, but it's usually fairly simple to use in practice. For example, most of the `CREATE TABLE` statements that we used in Chapter 1, "Getting Started with MySQL," are reasonably uncomplicated. If you start with the more basic forms and work up, you shouldn't have much trouble.

A `CREATE TABLE` statement specifies, at a minimum, the table name and a list of the columns in it. For example:

```
CREATE TABLE mytbl
(
  name   CHAR(20),
  birth  DATE NOT NULL,
  weight INT,
  sex    ENUM('F','M')
);
```

In addition to the column definitions, you can specify how the table should be indexed when you create it. Another option is to leave the table unindexed when you create it and add the indexes later. For MyISAM tables, that's a good strategy if you plan to populate the table with a lot of data before you begin using it for queries. Updating indexes as you insert each row is much slower than loading the data into an unindexed MyISAM table and creating the indexes afterward.

We have already covered the basic syntax for `CREATE TABLE` in Chapter 1, "Getting Started with MySQL." Details on how to write column definitions are given in Chapter 3, "Data Types." Here, we deal more generally with some important extensions to `CREATE TABLE` that give you a lot of flexibility in how you construct tables:

- Table options that modify storage characteristics
- Creating a table only if it doesn't already exist
- Temporary tables that the server drops automatically when the client session ends

- Creating a table from another table or from the result of a `SELECT` query
- Using partitioned tables

2.6.2.1 Table Options

To modify a table's storage characteristics, add one or more table options following the closing parenthesis in the `CREATE TABLE` statement. For a complete list of options, see the description for `CREATE TABLE` in Appendix E, "SQL Syntax Reference."

One table option is `ENGINE = engine_name`, which specifies the storage engine to use for the table. For example, to create a MEMORY or MyISAM table, write the statement like this:

```
CREATE TABLE mytbl ( ... ) ENGINE=MEMORY;
CREATE TABLE mytbl ( ... ) ENGINE=MyISAM;
```

The engine name is not case sensitive. With no `ENGINE` option, the server creates the table using the default storage engine. The built-in default is InnoDB, but you can tell the server to use a different default using the instructions in Section 12.5.2, "Selecting a Default Storage Engine."

If you name a storage engine that is not enabled, two warnings occur:

```
mysql> CREATE TABLE t (i INT) ENGINE=ARCHIVE;
Query OK, 0 rows affected, 2 warnings (0.01 sec)
mysql> SHOW WARNINGS;
+---------+------+------------------------------------------+
| Level   | Code | Message                                  |
+---------+------+------------------------------------------+
| Warning | 1286 | Unknown storage engine 'ARCHIVE'         |
| Warning | 1266 | Using storage engine InnoDB for table 't' |
+---------+------+------------------------------------------+
```

To make sure that a table uses a particular storage engine, be sure to include the `ENGINE` table option. Because the default engine can be changed, you might not get the default you expect if you omit `ENGINE`. In addition, verify that the `CREATE TABLE` statement produces no warnings, which often indicate that the specified engine was not available and that the default engine was used instead.

To tell MySQL to issue an error if the engine you specify is not available, (instead of substituting the default storage engine), enable the `NO_ENGINE_SUBSTITUTION` SQL mode.

To determine which storage engine a table uses, issue a `SHOW CREATE TABLE` statement and look for the `ENGINE` option in the output:

```
mysql> SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `i` int(11) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

The storage engine is also available in the output from the `SHOW TABLE STATUS` statement or the `INFORMATION_SCHEMA.TABLES` table.

The `MAX_ROWS` and `AVG_ROW_LENGTH` options can help you size a MyISAM table. By default, MyISAM creates tables with an internal row pointer size that permits table files to grow up to 256TB. If you specify the `MAX_ROWS` and `AVG_ROW_LENGTH` options, that gives MyISAM information that it should use a pointer size for a table that can hold at least `MAX_ROWS` rows.

To modify the storage characteristics of an existing table, table options can be used with an `ALTER TABLE` statement. For example, to change `mytbl` from its current storage engine to InnoDB, do this:

```
ALTER TABLE mytbl ENGINE=InnoDB;
```

For more information about changing storage engines, see Section 2.6.5, "Altering Table Structure."

2.6.2.2 Provisional Table Creation

To create a table only if it doesn't already exist, use `CREATE TABLE IF NOT EXISTS`. You can use this statement for an application that makes no assumptions about whether a table that it needs has been set up in advance. The application can go ahead and attempt to create the table as a matter of course. The `IF NOT EXISTS` modifier is particularly useful for scripts that you run as batch jobs with `mysql`. In this context, a regular `CREATE TABLE` statement doesn't work very well. The first time the job runs, it creates the table, but the second time, an error occurs because the table already exists. If you use `IF NOT EXISTS`, there is no problem. The first time the job runs, it creates the table, as before. For second and subsequent times, table creation attempts are silently ignored without error. This enables the job to continue processing as if the attempt had succeeded.

If you use `IF NOT EXISTS`, be aware that MySQL does not compare the table structure in the `CREATE TABLE` statement with the existing table. If a table exists with the given name but has a different structure, the statement does not fail. If that is a risk you wish not to take, it is better instead to precede your `CREATE TABLE` statement by `DROP TABLE IF EXISTS`.

2.6.2.3 `TEMPORARY` Tables

Adding the `TEMPORARY` keyword to a table-creation statement causes the server to create a temporary table that disappears automatically when your session with the server terminates:

```
CREATE TEMPORARY TABLE tbl_name ... ;
```

This is handy because you need not issue a `DROP TABLE` statement to get rid of the table, and the table doesn't persist if your session terminates abnormally. For example, if you have a complex query stored in a batch file that you run with `mysql` and you decide not to wait for it to finish, you can kill the script with impunity and the server will remove any `TEMPORARY` tables created by the script.

To create a temporary table using a particular storage engine, add an `ENGINE` table option to the `CREATE TEMPORARY TABLE` statement.

Although the server drops a `TEMPORARY` table automatically when your client session ends, you can drop it explicitly as soon as you're done with it to enable the server to free any resources associated with it. This is a good idea if your session with the server will not end for a while, particularly for temporary MEMORY tables.

A `TEMPORARY` table is visible only to the client that creates the table. Different clients can each create a `TEMPORARY` table with the same name and without conflict because each client sees only the table that it created.

The name of a `TEMPORARY` table can be the same as an existing permanent table. This is not an error, nor does the existing permanent table get clobbered. Instead, the permanent table becomes hidden (inaccessible) to the client that creates the `TEMPORARY` table while the `TEMPORARY` table exists. Suppose that you create a `TEMPORARY` table named `member` in the `sampdb` database. The original `member` table becomes hidden, and references to `member` refer to the `TEMPORARY` table. If you issue a `DROP TABLE member` statement, the `TEMPORARY` table is removed and the original member table "reappears." If you disconnect from the server without dropping the `TEMPORARY` table, the server automatically drops it for you. The next time you connect, the original `member` table is visible again. (The original table also reappears if you rename a `TEMPORARY` table that hides it to have a different name.)

The name-hiding mechanism works only to one level. That is, you cannot create two `TEMPORARY` tables with the same name.

Keep in mind the following caveats when considering whether to use a `TEMPORARY` table:

- If your client program automatically reconnects to the server if the connection is lost, any `TEMPORARY` tables will be gone when you reconnect. If you were using the `TEMPORARY` table to "hide" a permanent table with the same name, the permanent table now becomes the table that you use. For example, a `DROP TABLE` after an undetected reconnect will drop the permanent table. To avoid this problem, use `DROP TEMPORARY TABLE` instead.
- Because `TEMPORARY` tables are visible only within the session that created them, they are not useful with connection pooling mechanisms that do not guarantee the same connection for each statement that you issue.
- With connection pooling or persistent connections, your connection to the MySQL server will not necessarily close when your application terminates. Those mechanisms might hold the connection open for use by other clients, which means that you cannot assume that `TEMPORARY` tables will disappear automatically when your application terminates.

2.6.2.4 Creating Tables from Other Tables or Query Results

It's sometimes useful to create a copy of a table. For example, you might have a data file that you want to load into a table using `LOAD DATA`, but you're not quite sure about the options for specifying the data format. You can end up with malformed rows in the original table if you don't get the options right the first time. Using an empty copy of the original table enables you to experiment with the LOAD DATA options for specifying column and line delimiters until you're satisfied your input rows are being interpreted properly. Then you can load the file into the original table by rerunning the LOAD DATA statement with the original table name.

It's also sometimes desirable to save the result of a query into a table rather than displaying it on your screen. By saving the result, you can refer to it later without rerunning the original query, perhaps to perform further analysis on it.

MySQL provides two statements for creating new tables from other tables or from query results. These statements have differing advantages and disadvantages:

- CREATE TABLE … LIKE creates a new table as an empty copy of the original one. It copies the original table structure exactly, so that each column is preserved with all of its attributes. The index structure is copied as well. However, the new table is empty, so to populate it a second statement is needed (such as INSERT INTO … SELECT). Also, CREATE TABLE … LIKE cannot create a new table from a subset of the original table's columns, and it cannot use columns from any other table but the original one.
- CREATE TABLE … SELECT creates a new table from the result of an arbitrary SELECT statement. By default, this statement does not copy all column attributes such as AUTO_INCREMENT. Nor does creating a table by selecting data into it automatically copy any indexes from the original table, because result sets are not themselves indexed. On the other hand, CREATE TABLE … SELECT can both create and populate the new table in a single statement. It also can create a new table using a subset of the original table and include columns from other tables or columns created as the result of expressions.

To use CREATE TABLE … LIKE for creating an empty copy of an existing table, write a statement like this:

```
CREATE TABLE new_tbl_name LIKE tbl_name;
```

To create an empty copy of a table and then populate it from the original table, use CREATE TABLE … LIKE followed by INSERT INTO … SELECT:

```
CREATE TABLE new_tbl_name LIKE tbl_name;
INSERT INTO new_tbl_name SELECT * FROM tbl_name;
```

To create a table as a temporary copy of itself, include the TEMPORARY keyword:

```
CREATE TEMPORARY TABLE tbl_name LIKE tbl_name;
INSERT INTO tbl_name SELECT * FROM tbl_name;
```

Using a TEMPORARY table with the same name as the original can be useful when you want to try some statements that modify the contents of a table, without changing the original table. To use prewritten scripts that use the original table name, you need not edit them to refer to a different table. Just add the CREATE TEMPORARY TABLE and INSERT statements to the beginning of the script. The script will create a temporary copy and operate on the copy, which the server deletes when the script finishes. (However, bear in mind the auto-reconnect caveat noted in Section 2.6.2.3, "TEMPORARY Tables.")

To insert into the new table only some of the rows from the original table, add a WHERE clause that identifies which rows to select. The following statements create a new table named student_f that contains only the rows for female students from the student table:

```
CREATE TABLE student_f LIKE student;
INSERT INTO student_f SELECT * FROM student WHERE sex = 'f';
```

If you don't care about retaining the exact column definitions from the original table, CREATE TABLE ... SELECT sometimes is easier to use than CREATE TABLE ... LIKE because it can create and populate the new table in a single statement:

```
CREATE TABLE student_f SELECT * FROM student WHERE sex = 'f';
```

CREATE TABLE ... SELECT also can create new tables that don't contain exactly the same set of columns in an existing table. You can use it to cause a new table to spring into existence on the fly to hold the result of an arbitrary SELECT query. This makes it exceptionally easy to create a table fully populated with the data in which you're interested, ready to be used in further statements. However, the new table can contain strange column names if you're not careful. When you create a table by selecting data into it, the column names are taken from the columns that you are selecting. If a column is calculated as the result of an expression, the name of the column is the text of the expression, which creates a table with an unusual column name:

```
mysql> CREATE TABLE mytbl SELECT PI() * 2;
mysql> SELECT * FROM mytbl;
+----------+
| PI() * 2 |
+----------+
| 6.283185 |
+----------+
```

That's unfortunate, because the column name can be referred to directly only as a quoted identifier:

```
mysql> SELECT `PI() * 2` FROM mytbl;
+----------+
| PI() * 2 |
+----------+
| 6.283185 |
+----------+
```

To avoid this problem, use a column alias to provide a name that is easier to work with:

```
mysql> DROP TABLE mytbl;
mysql> CREATE TABLE mytbl SELECT PI() * 2 AS mycol;
mysql> SELECT mycol FROM mytbl;
+----------+
| mycol    |
+----------+
| 6.283185 |
+----------+
```

A related difficulty occurs if you select from different tables columns that have the same name. Suppose that tables `t1` and `t2` both have a column `c` and you want to create a table from all combinations of rows in both tables. The following statement fails because it attempts to create a table with two columns named `c`:

```
mysql> CREATE TABLE t3 SELECT * FROM t1 INNER JOIN t2;
ERROR 1060 (42S21): Duplicate column name 'c'
```

To solve this problem, provide aliases as necessary to give each column a unique name in the new table:

```
mysql> CREATE TABLE t3 SELECT t1.c, t2.c AS c2
    -> FROM t1 INNER JOIN t2;
```

As mentioned previously, a shortcoming of `CREATE TABLE ... SELECT` is that it does not incorporate all characteristics of the original data into the structure of the new table. For example, creating a table by selecting data into it does not copy indexes from the original table, and it can lose column attributes. The retained attributes include whether the column is `NULL` or `NOT NULL`, the character set and collation, the default value, and the column comment.

In some cases, you can force specific attributes to be used in the new table by invoking the `CAST()` function in the `SELECT` part of the statement. The following `CREATE TABLE ... SELECT` statement forces the columns produced by the `SELECT` to be treated as `INT UNSIGNED`, `TIME`, and `DECIMAL(10,5)`, as you can verify with `DESCRIBE`:

```
mysql> CREATE TABLE mytbl SELECT
    -> CAST(1 AS UNSIGNED) AS i,
    -> CAST(CURTIME() AS TIME) AS t,
    -> CAST(PI() AS DECIMAL(10,5)) AS d;
mysql> DESCRIBE mytbl;
+-------+-----------------+------+-----+---------+-------+
| Field | Type            | Null | Key | Default | Extra |
+-------+-----------------+------+-----+---------+-------+
| i     | int(1) unsigned | NO   |     | 0       |       |
| t     | time            | YES  |     | NULL    |       |
| d     | decimal(10,5)   | NO   |     | 0.00000 |       |
+-------+-----------------+------+-----+---------+-------+
```

The permitted cast types are `BINARY` (binary string), `CHAR`, `DATE`, `DATETIME`, `TIME`, `SIGNED`, `SIGNED INTEGER`, `UNSIGNED`, `UNSIGNED INTEGER`, and `DECIMAL`.

It is also possible to provide explicit column definitions in the `CREATE TABLE` part, to be used for the columns retrieved by the `SELECT` part. Columns in the two parts are matched by name (not position), so provide aliases in the `SELECT` part as necessary to cause them to match properly:

```
mysql> CREATE TABLE mytbl (i INT UNSIGNED, t TIME, d DECIMAL(10,5))
    -> SELECT
    -> 1 AS i,
    -> CAST(CURTIME() AS TIME) AS t,
```

```
    -> CAST(PI() AS DECIMAL(10,5)) AS d;
mysql> DESCRIBE mytbl;
+-------+------------------+------+-----+---------+-------+
| Field | Type             | Null | Key | Default | Extra |
+-------+------------------+------+-----+---------+-------+
| i     | int(10) unsigned | YES  |     | NULL    |       |
| t     | time             | YES  |     | NULL    |       |
| d     | decimal(10,5)    | YES  |     | NULL    |       |
+-------+------------------+------+-----+---------+-------+
```

The technique of providing explicit definitions enables you to create numeric columns with specified precision and scale, character columns that have a different width than the longest value in the result set, and so forth. Also note that the `Null` and `Default` attributes for some of the columns differ in this example from those in the previous one. You can provide explicit definitions for those attributes in the `CREATE TABLE` part if necessary.

2.6.2.5 Using Partitioned Tables

MySQL supports table partitioning, which enables division of table contents into different physical storage locations. By sectioning table storage, partitioned tables offer benefits such as these:

- Table storage can be distributed over multiple devices, which may improve access time by virtue of I/O parallelism.
- The optimizer may be able to localize searches to specific partitions, or to search partitions in parallel.

To create a partitioned table, supply the list of columns and indexes in the `CREATE TABLE` statement, as usual. In addition, specify a `PARTITION BY` clause that defines a partitioning function to be used to assign rows to partitions, and possibly other partition-related options. A partitioning function assigns rows based on ranges or lists of values or hash values:

- Use range partitioning when rows contain a domain of values such as dates, income level, or weight that can be divided into discrete ranges.
- Use list partitioning when it makes sense to specify an explicit list of values for each partition, such as sets of postal codes, phone number prefixes, or IDs for entities that you group by geographical region.
- Use hash partitioning to distribute the rows among partitions according to hash values computed from row keys. You can either supply the hash function yourself or tell MySQL which columns to use and it computes values based on those columns using a built-in hash function.

The partitioning function must be deterministic so that the same input values consistently result in row assignment to the same partition. This rules out functions such as `RAND()` or `NOW()`.

Suppose that you want to create a table for storing simple log entries consisting of a date and a descriptive string, and that you already have several years' worth of entries to be loaded into the table. For data entries that each contain a date, range partitioning is most natural. To assign rows for each year to a given partition, use the year part of the date value:

```
CREATE TABLE log_partition
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
)
PARTITION BY RANGE(YEAR(dt))
(
  PARTITION p0 VALUES LESS THAN (2010),
  PARTITION p1 VALUES LESS THAN (2011),
  PARTITION p2 VALUES LESS THAN (2012),
  PARTITION p3 VALUES LESS THAN (2013),
  PARTITION pmax VALUES LESS THAN MAXVALUE
);
```

The `MAXVALUE` partition is assigned all rows that have dates from the year 2014 or later. When the year 2014 arrives, you can split that partition so that all year 2014 rows get their own partition and rows for 2015 and later go into the `MAXVALUE` partition:

```
ALTER TABLE log_partition REORGANIZE PARTITION pmax
INTO (
  PARTITION p4 VALUES LESS THAN (2014),
  PARTITION pmax VALUES LESS THAN MAXVALUE
);
```

By default, MySQL stores partitions under the directory for the database to which the partitioned table belongs. To distribute storage to other locations (for example, to place them on different physical devices), use the `DATA_DIRECTORY` and `INDEX_DIRECTORY` partition options. For more information about the syntax for these and other partitioning options, see the description for `CREATE TABLE` in Appendix E, "SQL Syntax Reference."

2.6.3 Dropping Tables

Dropping a table is much easier than creating it because you need not specify anything about the format of its contents. You just have to name it, assuming that you have the `DROP` privilege for it:

```
DROP TABLE tbl_name;
```

In MySQL, the `DROP TABLE` statement has several useful extensions. To drop multiple tables, specify them all in the same statement:

```
DROP TABLE tbl_name1, tbl_name2, ... ;
```

By default, an error occurs if you try to drop a table that does not exist. To suppress this error and generate a warning instead for nonexistent tables, include `IF EXISTS` in the statement:

```
DROP TABLE IF EXISTS tbl_name;
```

If the statement generates warnings, you can view them with `SHOW WARNINGS`.

`IF EXISTS` is particularly useful in scripts that you use with the `mysql` client. By default, `mysql` exits when an error occurs, and it is an error to try to remove a table that doesn't exist. For example, you might have a setup script that creates tables used as the basis for further processing in other scripts. In this situation, you want to make sure the setup script has a clean slate when it begins. If you use a regular `DROP TABLE` at the beginning of the script, it fails the first time because the tables have never been created. Using `IF EXISTS` makes the problem go away. If the tables exist, they are dropped. If they do not exist, no error occurs and the script continues to execute.

To drop a table only if it is a temporary table, include the `TEMPORARY` keyword:

```
DROP TEMPORARY TABLE tbl_name;
```

2.6.4 Indexing Tables

Indexes are the primary means of speeding up access to the contents of your tables, particularly for queries that involve joins on multiple tables. This is an important enough topic that most of an entire chapter discusses why you use indexes, how they work, and how best to take advantage of them to optimize your queries (see Chapter 5, "Query Optimization"). This section covers the characteristics of indexes for the various table types and the syntax for creating and dropping indexes.

2.6.4.1 Storage Engine Index Characteristics

MySQL provides quite a bit of flexibility for index construction:

- You can index single columns or multiple columns. Multiple-column indexes are also known as composite indexes.
- An index can be constrained to contain only unique values or permitted to contain duplicate values.
- You can have more than one index on a table to help optimize different types of queries on the table.
- For string data types other than `ENUM` or `SET`, you can elect to index a prefix of a column; that is, only the leftmost *n* characters, or *n* bytes for binary string types. (For `BLOB` and `TEXT` columns, you can set up an index only if you specify a prefix length.) If the column is mostly unique within the prefix length, you usually won't sacrifice performance, and may well improve it: Indexing a column prefix rather than the entire column can make an index much smaller and faster to access.

Not all storage engines offer all indexing features. Table 2.3 summarizes the index properties for some of MySQL's storage engines. The table does not include the MERGE storage engine, because MERGE tables are created from MyISAM tables and have similar index characteristics. Nor does it include the ARCHIVE, BLACKHOLE, or CSV engines, which support indexing either not at all or only in limited fashion.

Table 2.3 **Storage Engine Index Characteristics**

Index Characteristic	InnoDB	MyISAM	MEMORY
`NULL` values permitted	Yes	Yes	Yes
Columns per index	16	16	16
Indexes per table	64	64	64
Maximum index row size (bytes)	3072	1000	3072
Index column prefixes	Yes	Yes	Yes
Maximum prefix size (bytes)	767	1000	3072
`BLOB`/`TEXT` indexes	Yes	Yes	No
`FULLTEXT` indexes	As of 5.6.4	Yes	No
`SPATIAL` indexes	No	Yes	No
`HASH` indexes	No	No	Yes

One implication of the variations in index characteristics for different storage engines is that if you require an index to have certain properties, you may not be able to use certain types of tables. For example, to use a `HASH` index, you must use a MEMORY table. To index a `TEXT` column, you must use InnoDB or MyISAM.

To convert an existing table to use a different storage engine that has more suitable index characteristics, use `ALTER TABLE`. Suppose that you have an InnoDB table in MySQL 5.5 but need to perform searches using a `FULLTEXT` index. In MySQL 5.5, this is supported only by MyISAM. Convert the table using this statement:

```
ALTER TABLE tbl_name ENGINE=MyISAM;
```

2.6.4.2 Creating Indexes

MySQL can create several types of indexes:

- A unique index. This prohibits duplicate values for a single-column index, and duplicate combinations of values for a multiple-column (composite) index.
- A regular (nonunique) index. This gives you indexing benefits but permits duplicates.

- A `FULLTEXT` index, used for performing full-text searches. This index type is supported only for MyISAM tables (or, as of MySQL 5.6.4, InnoDB). For more information, see Section 2.14, "Using `FULLTEXT` Searches."
- A `SPATIAL` index. These can be used only with MyISAM tables containing spatial values, which are described briefly in Section 3.1.4, "Spatial Values."
- A `HASH` index. This is the default index type for `MEMORY` tables, although you can override the default to create `BTREE` indexes instead.

You can include index definitions for a new table when you use `CREATE TABLE`. For examples, see Section 1.4.6, "Creating Tables." To add indexes to existing tables, use `ALTER TABLE` or `CREATE INDEX`. (MySQL maps `CREATE INDEX` statements onto `ALTER TABLE` operations internally.)

`ALTER TABLE` is the more versatile than `CREATE INDEX` because it can create any kind of index supported by MySQL. For example:

```
ALTER TABLE tbl_name ADD INDEX index_name (index_columns);
ALTER TABLE tbl_name ADD UNIQUE index_name (index_columns);
ALTER TABLE tbl_name ADD PRIMARY KEY (index_columns);
ALTER TABLE tbl_name ADD FULLTEXT index_name (index_columns);
ALTER TABLE tbl_name ADD SPATIAL index_name (index_columns);
```

`tbl_name` is the name of the table to which the index should be added, and *`index_columns`* names the column or columns to index, separated by commas. The index name *`index_name`* is optional. If you leave it out, MySQL picks a name based on the name of the first indexed column.

An indexed column must be `NOT NULL` if indexed using a `PRIMARY KEY` or `SPATIAL` index. Other indexes permit indexed columns to contain `NULL` values.

A single `ALTER TABLE` statement can include multiple table alterations if you separate them by commas. This enables you to create several indexes at the same time, which is faster than adding them one at a time with individual `ALTER TABLE` statements.

To constrain an index to contain only unique values, create the index as a `PRIMARY KEY` or as a `UNIQUE` index. The two types of index are very similar, but have two differences:

- A table can contain only one `PRIMARY KEY`. This is because the name of a `PRIMARY KEY` is always `PRIMARY` and a table cannot have two indexes with the same name. You can place multiple `UNIQUE` indexes on a table.
- A `PRIMARY KEY` cannot contain `NULL` values. A `UNIQUE` index can. If a `UNIQUE` index can contain `NULL` values, it can contain multiple `NULL` values. (A `NULL` is not considered equal to any other value, even another `NULL`.)

`CREATE INDEX` can add most types of indexes, with the exception of `PRIMARY KEY`:

```
CREATE INDEX index_name ON tbl_name (index_columns);
CREATE UNIQUE INDEX index_name ON tbl_name (index_columns);
```

```
CREATE FULLTEXT INDEX index_name ON tbl_name (index_columns);
CREATE SPATIAL INDEX index_name ON tbl_name (index_columns);
```

tbl_name, *index_name*, and *index_columns* have the same meaning as for `ALTER TABLE`. Unlike `ALTER TABLE`, the index name is not optional with `CREATE INDEX`, and you cannot create multiple indexes with a single statement.

To create indexes for a new table with a `CREATE TABLE` statement, the syntax is similar to that used for `ALTER TABLE`, but you specify the index-creation clauses in addition to the column definitions:

```
CREATE TABLE tbl_name
(
  ... column definitions ...
  INDEX index_name (index_columns),
  UNIQUE index_name (index_columns),
  PRIMARY KEY (index_columns),
  FULLTEXT index_name (index_columns),
  SPATIAL index_name (index_columns),
  ...
);
```

As with `ALTER TABLE`, *index_name* is optional. MySQL picks an index name if you leave it out.

As a special case, you can create a single-column `PRIMARY KEY` or `UNIQUE` index by adding a `PRIMARY KEY` or `UNIQUE` clause to the end of a column definition. For example, the following `CREATE TABLE` statements are equivalent:

```
CREATE TABLE mytbl
(
  i INT NOT NULL PRIMARY KEY,
  j CHAR(10) NOT NULL UNIQUE
);

CREATE TABLE mytbl
(
  i INT NOT NULL,
  j CHAR(10) NOT NULL,
  PRIMARY KEY (i),
  UNIQUE (j)
);
```

The default index type for a MEMORY table is `HASH`. A hashed index is very fast for exact-value lookups, which is the typical way MEMORY tables are used. However, if you plan to use a MEMORY table for comparisons that can match a range of values (for example, `id < 100`), hashed indexes do not work well. You'll be better off creating a `BTREE` index instead, by adding a `USING BTREE` clause to the index definition:

```
CREATE TABLE namelist
(
  id   INT NOT NULL,
  name CHAR(100),
  INDEX (id) USING BTREE
) ENGINE=MEMORY;
```

To index a prefix of a string column, the syntax for naming the column in the index definition is `col_name(n)` rather than simply `col_name`. The prefix value, *n*, indicates that the index should include the first *n* bytes of column values for binary string types, or the first *n* characters for nonbinary string types. For example, the following statement creates a table with a `CHAR` column and a `BINARY` column. It indexes the first 10 characters of the `CHAR` column and the first 15 bytes of the `BINARY` column:

```
CREATE TABLE addresslist
(
  name    CHAR(30) NOT NULL,
  address BINARY(60) NOT NULL,
  INDEX (name(10)),
  INDEX (address(15))
);
```

When you index a prefix of a string column, the prefix length, just like the column length, is specified in the same units as the column data type—that is, bytes for binary strings and characters for nonbinary strings. However, the maximum size of index entries are measured internally in bytes. The two measures are the same for single-byte character sets, but not for multi-byte character sets. For nonbinary strings that have multi-byte character sets, MySQL stores into index values as many complete characters as fit within the maximum permitted byte length.

In some circumstances, you may find it not only desirable but necessary to index a column prefix rather than the entire column:

- A prefix is required to index a `BLOB` or `TEXT` column.
- The length of index rows is equal to the sum of the length of the index parts of the columns that make up the index. If this length exceeds the maximum permitted number of bytes in index rows, you can make the index "narrower" by indexing a column prefix. Suppose that a MyISAM table that uses the `latin1` single-byte character set contains four `CHAR(255)` columns named `c1` through `c4`. An index value for each full column value takes 255 bytes, so an index on all four columns would require 1,020 bytes. However, the maximum length of a MyISAM index row is 1,000 bytes, so you cannot create a composite index that includes the entire contents of all four columns. However, you can create the index by indexing a shorter part of some or all of them. For example, you could index the first 250 characters from each column.

Columns in `FULLTEXT` indexes are indexed in full and do not have prefixes. If you specify a prefix length for a column in a `FULLTEXT` index, MySQL ignores it.

2.6.4.3 Dropping Indexes

To drop an index, use either a `DROP INDEX` or an `ALTER TABLE` statement. To use `DROP INDEX`, you must name the index to be dropped:

```
DROP INDEX index_name ON tbl_name;
```

To drop a `PRIMARY KEY` with `DROP INDEX`, specify the name `PRIMARY` as a quoted identifier:

```
DROP INDEX `PRIMARY` ON tbl_name;
```

That statement is unambiguous because a table is permitted only one `PRIMARY KEY` and its name is always `PRIMARY`.

Like the `CREATE INDEX` statement, `DROP INDEX` is handled internally as an `ALTER TABLE` statement. The preceding `DROP INDEX` statements correspond to the following `ALTER TABLE` statements:

```
ALTER TABLE tbl_name DROP INDEX index_name;
ALTER TABLE tbl_name DROP PRIMARY KEY;
```

If you don't know the names of a table's indexes, use `SHOW CREATE TABLE` or `SHOW INDEX` to find out.

When you drop columns from a table, indexes may be affected implicitly. Dropping a column that is a part of an index removes the column from the index as well. If you drop all columns in an index, MySQL drops the entire index.

2.6.5 Altering Table Structure

`ALTER TABLE` is a versatile statement and has many uses. We've already seen a few of its capabilities earlier in this chapter (for changing storage engines and for creating and dropping indexes). `ALTER TABLE` can also rename tables, add or drop columns, change column data types, and more. This section covers some of its features. For its complete syntax, see Appendix E, "SQL Syntax Reference."

`ALTER TABLE` is useful when you find that the structure of a table no longer reflects its intended use. Perhaps you want to record additional information, or the table contains information that has become superfluous. Maybe existing columns are too small, or it turns out that you've defined columns larger than you need and you'd like to make them smaller to save space and improve query performance. Here are some situations for which `ALTER TABLE` is valuable:

- You assign case numbers to records for a research project using an `AUTO_INCREMENT` column. You didn't expect your funding to last long enough to generate more than about 50,000 records, so you made the data type `SMALLINT UNSIGNED`, which holds a maximum of 65,535 unique values. However, the funding for the project was renewed, and it looks like you might generate another 50,000 records. You need a bigger type to accommodate more case numbers.

- Size changes can go the other way, too. Maybe you created a `CHAR(255)` column but now recognize that no value in the table is more than 100 characters long. You can shorten the column or convert it to `VARCHAR(255)` to save space.
- You want to convert a table to use a different storage engine to take advantage of features offered by that engine. For example, MyISAM tables are not transaction-safe, but you have an application that needs transactional capabilities. You can convert the affected tables to use InnoDB, which supports transactions. Or you might be using MyISAM in MySQL 5.5 because it supports `FULLTEXT` capabilities, but now you have upgraded to MySQL 5.6, which expands `FULLTEXT` support to InnoDB.

The syntax for `ALTER TABLE` looks like this:

```
ALTER TABLE tbl_name action [, action] ... ;
```

Each action specifies a modification to make to the table. Some database systems permit only a single action in an `ALTER TABLE` statement, but MySQL supports multiple actions, separated by commas.

> **Tip**
>
> If you need to remind yourself about a table's current definition before using `ALTER TABLE`, issue a `SHOW CREATE TABLE` statement. This statement is also useful after `ALTER TABLE` to verify that the alteration affected the table definition as you expect.

The following examples discuss some of the capabilities of `ALTER TABLE`.

Changing a column's data type. To change a data type, use either a `CHANGE` or `MODIFY` clause. Suppose that the column `i` in a table `mytbl` is `SMALLINT UNSIGNED`. To change it to `MEDIUMINT UNSIGNED`, use either of the following statements:

```
ALTER TABLE mytbl MODIFY i MEDIUMINT UNSIGNED;
ALTER TABLE mytbl CHANGE i i MEDIUMINT UNSIGNED;
```

Why is the column named twice in the statement that uses `CHANGE`? Because one thing that `CHANGE` can do that `MODIFY` cannot is to rename the column in addition to changing the type. If you had wanted to rename `i` to `k` at the same time you changed the type, you'd do so like this:

```
ALTER TABLE mytbl CHANGE i k MEDIUMINT UNSIGNED;
```

Remember that with `CHANGE`, you name the column you want to change and then specify its new name and definition. To retain the same column name, you must specify the name twice.

To rename a column without changing its data type, use `CHANGE` *`old_name new_name`* followed by the column's current definition.

To change a column's character set, use the `CHARACTER SET` attribute in the column definition:

```
ALTER TABLE t MODIFY c CHAR(20) CHARACTER SET ucs2;
```

An important reason for changing data types is to improve query efficiency for joins that compare columns from two tables. Indexes often can be used for comparisons in joins between similar column types, but comparisons are quicker when both columns are exactly the same type. Suppose that you're running a query like this:

```
SELECT ... FROM t1 INNER JOIN t2 WHERE t1.name = t2.name;
```

If `t1.name` is `CHAR(10)` and `t2.name` is `CHAR(15)`, the query won't run as quickly as if they were both `CHAR(15)`. You can make them the same by changing `t1.name` using either of these statements:

```
ALTER TABLE t1 MODIFY name CHAR(15);
ALTER TABLE t1 CHANGE name name CHAR(15);
```

Converting a table to a different storage engine. To convert a table from one storage engine to another, use an `ENGINE` clause that specifies the new engine name:

```
ALTER TABLE tbl_name ENGINE=engine_name;
```

`engine_name` is a name such as `InnoDB`, `MyISAM`, or `MEMORY`. Lettercase does not matter.

One reason to change a storage engine is to make it transaction-safe. Suppose that you have a MyISAM table and discover that an application that uses it needs to perform transactional operations, including rollback in case failures occur. MyISAM tables do not support transactions, but you can make the table transaction-safe by converting it to use InnoDB:

```
ALTER TABLE tbl_name ENGINE=InnoDB;
```

When you convert a table to a different engine, the permitted or sensible conversions may depend on the feature compatibility of the old and new engines. For example, if you have a table that includes a `BLOB` column, you cannot convert the table to use the MEMORY engine because MEMORY tables do not support `BLOB` columns.

There are circumstances under which you should not use `ALTER TABLE` to convert a table to use a different storage engine. For example:

- An InnoDB table can be converted to use another storage engine. However, if the table has foreign key constraints, they will be lost because only InnoDB supports foreign keys.
- MEMORY tables are held in memory and disappear when the server exits. If you require a table's contents to persist across server restarts, do not convert it to use the MEMORY engine.

Renaming a table. Use a `RENAME` clause that specifies the new table name:

```
ALTER TABLE tbl_name RENAME TO new_tbl_name;
```

Another way to rename tables is with `RENAME TABLE`. The syntax looks like this:

```
RENAME TABLE tbl_name TO new_tbl_name;
```

One thing that RENAME TABLE can do that ALTER TABLE cannot is rename multiple tables in the same statement. For example, you can swap the names of two tables like this:

```
RENAME TABLE t1 TO tmp, t2 TO t1, tmp TO t2;
```

If you qualify a table name with a database name, you can move a table from one database to another by renaming it. Either of the following statements move the table t from the sampdb database to the test database:

```
ALTER TABLE sampdb.t RENAME TO test.t;
RENAME TABLE sampdb.t TO test.t;
```

You cannot rename a table to a name that already exists.

2.7 Obtaining Database Metadata

MySQL provides several ways to obtain database metadata—that is, information about databases and the objects in them:

- SHOW statements such as SHOW DATABASES or SHOW TABLES
- Tables in the INFORMATION_SCHEMA database
- Command-line programs such as mysqlshow or mysqldump

The following sections describe how to use each of these information sources to access metadata.

2.7.1 Obtaining Metadata with SHOW

MySQL provides a SHOW statement that displays many types of database metadata. SHOW is helpful for keeping track of the contents of your databases and reminding yourself about the structure of your tables. The following examples demonstrate a few uses for SHOW statements.

List the databases you can access:

```
SHOW DATABASES;
```

Display the CREATE DATABASE statement for a database:

```
SHOW CREATE DATABASE db_name;
```

List the tables in the default database or a given database:

```
SHOW TABLES;
SHOW TABLES FROM db_name;
```

SHOW TABLES doesn't show TEMPORARY tables.

Display the CREATE TABLE statement for a table:

```
SHOW CREATE TABLE tbl_name;
```

Display information about columns or indexes in a table:

```
SHOW COLUMNS FROM tbl_name;
SHOW INDEX FROM tbl_name;
```

The `DESCRIBE tbl_name` and `EXPLAIN tbl_name` statements are synonymous with `SHOW COLUMNS FROM tbl_name`.

Display descriptive information about tables in the default database or in a given database:

```
SHOW TABLE STATUS;
SHOW TABLE STATUS FROM db_name;
```

Several forms of the `SHOW` statement take a `LIKE 'pattern'` clause permitting a pattern to be given that limits the scope of the output. MySQL interprets `'pattern'` as an SQL pattern that may include the '`%`' and '`_`' wildcard characters. For example, this statement displays the names of columns in the `student` table that begin with '`s`':

```
mysql> SHOW COLUMNS FROM student LIKE 's%';
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| sex        | enum('F','M')    | NO   |     | NULL    |                |
| student_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

To match a literal instance of a wildcard character in a `LIKE` pattern, precede it with a backslash. This is commonly done to match a literal '_', which occurs frequently in database, table, and column names.

Any `SHOW` statement that supports a `LIKE` clause can also be written to use a `WHERE` clause. The statement displays the same columns, but `WHERE` provides more flexibility about specifying which rows to return. The `WHERE` clause should refer to the `SHOW` statement column names. If the column name is a reserved word such as `KEY`, specify it as a quoted identifier. This statement determines which column in the `student` table is the primary key:

```
mysql> SHOW COLUMNS FROM student WHERE `Key` = `PRI`;
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| student_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

It's sometimes useful to be able to tell from within an application whether a given table exists. You can use `SHOW TABLES` to find out (unless the table is a `TEMPORARY` table):

```
SHOW TABLES LIKE 'tbl_name';
SHOW TABLES FROM db_name LIKE 'tbl_name';
```

If the `SHOW TABLES` statement lists information for the table, it exists. It's also possible to determine table existence, even for `TEMPORARY` tables, with either of the following statements:

```
SELECT COUNT(*) FROM tbl_name;
SELECT * FROM tbl_name WHERE FALSE;
```

Each statement succeeds if the table exists, and fails if it doesn't. The first statement is most appropriate for MyISAM tables, for which `COUNT(*)` with no `WHERE` clause is highly optimized. It's not so good for InnoDB tables, which require a full scan to count the rows. The second statement is more general because it runs quickly for any storage engine. These statements are most suitable for use within application programming languages such as Perl or PHP because you can test the success or failure of the query and take action accordingly. They're not especially useful in a batch script that you run from `mysql` because you can't do anything if an error occurs except terminate (or ignore the error, but then there's obviously no point in running the query at all). Another strategy, which works in any context without failure, is to query the `INFORMATION_SCHEMA` database. See Section 2.7.2, "Obtaining Metadata with `INFORMATION_SCHEMA`."

To determine the storage engine for individual tables, you can use `SHOW TABLE STATUS` or `SHOW CREATE TABLE`. The output from either statement includes a storage engine indicator.

2.7.2 Obtaining Metadata with INFORMATION_SCHEMA

Another way to obtain information about databases is to access the `INFORMATION_SCHEMA` database. `INFORMATION_SCHEMA` is based on the SQL standard. That is, the access mechanism is standard, even though some of the content is MySQL-specific. This makes `INFORMATION_SCHEMA` more portable than the various `SHOW` statements, which are entirely MySQL-specific.

`INFORMATION_SCHEMA` is accessed through `SELECT` statements and can be used in a flexible manner. `SHOW` statements always display a fixed set of columns and you cannot capture the output in a table. With `INFORMATION_SCHEMA`, the `SELECT` statement can name specific output columns and a `WHERE` clause can specify any expression required to select the information that you want. Also, you can use joins or subqueries, and you can use `CREATE TABLE ... SELECT` or `INSERT INTO ... SELECT` to save the result of the retrieval in another table for further processing.

You can think of `INFORMATION_SCHEMA` as a virtual database in which the tables are views for different kinds of database metadata. To see what tables `INFORMATION_SCHEMA` contains, use `SHOW TABLES`:

```
mysql> SHOW TABLES IN INFORMATION_SCHEMA;
+---------------------------------------+
| Tables_in_information_schema          |
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                               |
| COLUMN_PRIVILEGES                     |
| ENGINES                               |
| EVENTS                                |
```

```
| FILES                                 |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| KEY_COLUMN_USAGE                      |
| PARAMETERS                            |
| PARTITIONS                            |
| PLUGINS                               |
| PROCESSLIST                           |
| PROFILING                             |
| REFERENTIAL_CONSTRAINTS               |
| ROUTINES                              |
| SCHEMATA                              |
| SCHEMA_PRIVILEGES                     |
| SESSION_STATUS                        |
| SESSION_VARIABLES                     |
| STATISTICS                            |
| TABLES                                |
| TABLESPACES                           |
| TABLE_CONSTRAINTS                     |
| TABLE_PRIVILEGES                      |
| TRIGGERS                              |
| USER_PRIVILEGES                       |
| VIEWS                                 |
+---------------------------------------+
```

The following list briefly describes some of the `INFORMATION_SCHEMA` tables just shown:

- `SCHEMATA`, `TABLES`, `VIEWS`, `ROUTINES`, `TRIGGERS`, `EVENTS`, `PARAMETERS`, `PARTITIONS`, `COLUMNS`

  Information about databases; tables, views, stored routines, triggers, and events within databases; routine parameters; table partitions; and columns within tables

- `FILES`

  Information about the files used to store tablespace data

- `TABLE_CONSTRAINTS`, `KEY_COLUMN_USAGE`

  Information about tables and columns that have constraints such as unique-valued indexes or foreign keys

- `STATISTICS`

  Information about table index characteristics

- `REFERENTIAL_CONSTRAINTS`

  Information about foreign keys

- `CHARACTER_SETS`, `COLLATIONS`, `COLLATION_CHARACTER_SET_APPLICABILITY`

  Information about supported character sets, collations for each character set, and mapping from each collation to its character set

- ENGINES, PLUGINS

  Information about storage engines and server plugins

- USER_PRIVILEGES, SCHEMA_PRIVILEGES, TABLE_PRIVILEGES, COLUMN_PRIVILEGES

  Global, database, table, and column privilege information from the user, db, tables_priv, and columns_priv tables in the mysql database

- GLOBAL_VARIABLES, SESSION_VARIABLES, GLOBAL_STATUS, SESSION_STATUS

  Global and session values of system and status variables

- PROCESSLIST

  Information about the threads executing within the server

Individual storage engines may add their own tables to INFORMATION_SCHEMA. For example, InnoDB does this.

To determine the columns contained in a given INFORMATION_SCHEMA table, use SHOW COLUMNS or DESCRIBE:

```
mysql> DESCRIBE INFORMATION_SCHEMA.CHARACTER_SETS;
+----------------------+-------------+------+-----+---------+-------+
| Field                | Type        | Null | Key | Default | Extra |
+----------------------+-------------+------+-----+---------+-------+
| CHARACTER_SET_NAME   | varchar(32) | NO   |     |         |       |
| DEFAULT_COLLATE_NAME | varchar(32) | NO   |     |         |       |
| DESCRIPTION          | varchar(60) | NO   |     |         |       |
| MAXLEN               | bigint(3)   | NO   |     | 0       |       |
+----------------------+-------------+------+-----+---------+-------+
```

To display information from a table, use a SELECT statement. (Neither INFORMATION_SCHEMA nor any of its table or column names are case sensitive.) The general query to see all the columns in any given INFORMATION_SCHEMA table is as follows:

```
SELECT * FROM INFORMATION_SCHEMA.tbl_name;
```

Include a WHERE clause to be specific about what you want to see.

The preceding section described the use of SHOW statements to determine whether a table exists or which storage engine it uses. INFORMATION_SCHEMA tables can provide the same information. This query uses INFORMATION_SCHEMA to test for the existence of a particular table, returning 1 or 0 to indicate that the table does or does not exist, respectively:

```
mysql> SELECT COUNT(*) FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA='sampdb' AND TABLE_NAME='member';
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
```

Use this query to check which storage engine a table uses:

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA='sampdb' AND TABLE_NAME='student';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+
```

2.7.3 Obtaining Metadata from the Command Line

The `mysqlshow` command provides some of the same information as certain `SHOW` statements, which enables you to get database and table information at your command prompt.

List databases managed by the server:

```
% mysqlshow
```

List tables in a database:

```
% mysqlshow db_name
```

Display information about columns in a table:

```
% mysqlshow db_name tbl_name
```

Display information about indexes in a table:

```
% mysqlshow --keys db_name tbl_name
```

Display descriptive information about tables in a database:

```
% mysqlshow --status db_name
```

The `mysqldump` client program enables you to see the structure of your tables in the form of a `CREATE TABLE` statement (much like `SHOW CREATE TABLE`). If you use `mysqldump` to review table structure, invoke it with the `--no-data` option so that you don't get swamped with your table's data!

```
% mysqldump --no-data db_name [tbl_name] ...
```

If you specify only the database name with no table names, `mysqldump` displays the structure for all tables in the database. Otherwise, it shows information only for the named tables.

For both `mysqlshow` and `mysqldump`, specify the usual connection parameter options as necessary, such as `--host`, `--user`, or `--password`.

2.8 Performing Multiple-Table Retrievals with Joins

It does no good to put records in a database unless you retrieve them eventually and do something with them. That's the purpose of the `SELECT` statement: to help you get at your data. `SELECT` probably is used more often than any other statement in the SQL language, but it can also be the trickiest; the conditions you use for choosing rows can be arbitrarily complex and can involve comparisons between columns in many tables.

The basic syntax of the `SELECT` statement looks like this:

```
SELECT select_list        # the columns to select
FROM table_list           # the tables from which to select rows
WHERE row_constraint      # the conditions rows must satisfy
GROUP BY grouping_columns # how to group results
ORDER BY sorting_columns  # how to sort results
HAVING group_constraint   # the conditions groups must satisfy
LIMIT count;              # row count limit on results
```

Everything in this syntax is optional except the word `SELECT` and the *`select_list`* part that specifies what you want to produce as output. Some databases require the `FROM` clause as well. MySQL does not, which enables you to evaluate expressions without referring to any tables:

```
SELECT SQRT(POW(3,2)+POW(4,2));
```

In Chapter 1, "Getting Started with MySQL," we devoted quite a bit of attention to single-table `SELECT` statements, concentrating primarily on the output column list and the `WHERE`, `GROUP BY`, `ORDER BY`, `HAVING`, and `LIMIT` clauses. This section covers an aspect of `SELECT` that is often confusing: writing joins; that is, `SELECT` statements that retrieve rows from multiple tables. We'll discuss the types of join MySQL supports, what they mean, and how to specify them. This should help you employ MySQL more effectively because, in many cases, the real problem of figuring out how to write a query is determining the proper way to join tables.

One problem with using `SELECT` is that when you first encounter a new type of problem, it's not always easy to see how to write a `SELECT` query to solve it. However, after you figure it out, you can use that experience when you run across similar problems in the future. `SELECT` is probably the statement for which past experience plays the largest role in being able to use it effectively, simply because of the sheer variety of problems to which it applies. As you gain experience, you'll be able to adapt joins more easily to new problems, and you'll find yourself thinking things like, "Oh, yes, that's one of those `LEFT JOIN` things," or, "Aha, that's a three-way join restricted by the common pairs of key columns." (You may find it encouraging to hear that experience helps you. Or you may find it alarming to consider that you could wind up thinking in terms like that.)

Many of the examples that demonstrate how to use the forms of join operations that MySQL supports use the following two tables, `t1` and `t2`:

```
Table t1:     Table t2:
+----+----+   +----+----+
| i1 | c1 |   | i2 | c2 |
```

```
+----+----+   +----+----+
|  1 | a  |   |  2 | c  |
|  2 | b  |   |  3 | b  |
|  3 | c  |   |  4 | a  |
+----+----+   +----+----+
```

The tables are deliberately small so the effect of each type of join can be readily seen.

Other types of multiple-table `SELECT` statement are subqueries (one `SELECT` nested within another) and `UNION` statements. These are covered in Section 2.9, "Performing Multiple-Table Retrievals with Subqueries," and Section 2.10, "Performing Multiple-Table Retrievals with `UNION`."

A related multiple-table feature that MySQL supports is the capability of deleting or updating rows in one table based on the contents of another. For example, you might want to remove rows in one table that aren't matched by any row in another, or copy values from columns in one table to columns in another. Section 2.11, "Multiple-Table Deletes and Updates," discusses these types of operations.

2.8.1 Inner Joins

If a `SELECT` statement names multiple tables in the `FROM` clause with the names separated by `INNER JOIN`, MySQL performs an inner join, which produces results by matching rows in one table with rows in another table. For example, if you join `t1` and `t2` as follows, each row in `t1` is combined with each row in `t2`:

```
mysql> SELECT * FROM t1 INNER JOIN t2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  1 | a  |  2 | c  |
|  2 | b  |  2 | c  |
|  3 | c  |  2 | c  |
|  1 | a  |  3 | b  |
|  2 | b  |  3 | b  |
|  3 | c  |  3 | b  |
|  1 | a  |  4 | a  |
|  2 | b  |  4 | a  |
|  3 | c  |  4 | a  |
+----+----+----+----+
```

In this statement, `SELECT *` means "select every column from every table named in the `FROM` clause." You could also write this as `SELECT t1.*, t2.*`:

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2;
```

If you don't want to select all columns or you want to display them in a different left-to-right order, name each desired column, separated by commas.

A join that combines each row of each table with each row in every other table to produce all possible combinations is known as the "cartesian product." Joining tables this way has the potential to produce a very large number of rows because the possible row count is the product of the number of rows in each table. A join between three tables that contain 100, 200, and 300 rows, respectively, could return 100 × 200 × 300 = 6 million rows. That's a lot of rows, even though the individual tables are small. In cases like this, normally a `WHERE` clause is useful for reducing the result set to a more manageable size.

If you add a `WHERE` clause causing tables to be matched on the values of certain columns, the join selects only rows with equal values in those columns:

```
mysql> SELECT t1.*, t2.* FROM t1 INNER JOIN t2 WHERE t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  2 | b  |  2 | c  |
|  3 | c  |  3 | b  |
+----+----+----+----+
```

The `CROSS JOIN` and `JOIN` join types are the same as `INNER JOIN`, so these statements are equivalent:

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 CROSS JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 JOIN t2 WHERE t1.i1 = t2.i2;
```

The '`,`' (comma) join operator is similar as well:

```
SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
```

However, the comma operator has a different precedence from the other join types, and it can sometimes produce syntax errors when the other types will not. I recommend that you avoid the comma operator.

`INNER JOIN`, `CROSS JOIN`, and `JOIN` (but not the comma operator) support alternative syntaxes for specifying how to match table columns:

- One syntax uses an `ON` clause rather than a `WHERE` clause. The following example shows this using `INNER JOIN`:

  ```
  SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ON t1.i1 = t2.i2;
  ```

  `ON` can be used regardless of whether the joined columns have the same name.
- The other syntax involves a `USING()` clause; this is similar in concept to `ON`, but the name of the joined column or columns must be the same in each table. For example, the following query joins `mytbl1.b` to `mytbl2.b`:

  ```
  SELECT mytbl1.*, mytbl2.* FROM mytbl1 INNER JOIN mytbl2 USING (b);
  ```

2.8.2 Qualifying References to Columns from Joined Tables

References to each table column throughout a `SELECT` statement must resolve unambiguously to a single table named in the `FROM` clause. If only one table is named, there is no ambiguity; all columns must be columns of that table. If multiple tables are named, any column name that appears in only one table is similarly unambiguous. However, if a column name appears in multiple tables, references to the column must be qualified with a table identifier using *`tbl_name.col_name`* syntax to specify which table you mean. Suppose that a table `mytbl1` contains columns `a` and `b`, and a table `mytbl2` contains columns `b` and `c`. References to columns `a` or `c` are unambiguous, but references to `b` must be qualified as either `mytbl1.b` or `mytbl2.b`:

```
SELECT a, mytbl1.b, mytbl2.b, c FROM mytbl1 INNER JOIN mytbl2 ... ;
```

Sometimes a table name qualifier is not sufficient to resolve a column reference. For example, if you're performing a self-join (that is, joining a table to itself), you're using the table multiple times within the query and it doesn't help to qualify a column name with the table name. In this case, table aliases are useful for communicating your intent. You can assign an alias to any instance of the table and refer to columns from that instance as *`alias_name.col_name`*. The following query joins a table to itself, but assigns an alias to one instance of the table to enable column references to be specified unambiguously:

```
SELECT mytbl.col1, m.col2 FROM mytbl INNER JOIN mytbl AS m
WHERE mytbl.col1 > m.col1;
```

2.8.3 Left and Right (Outer) Joins

An inner join shows only rows where a match can be found in both tables. An outer join shows matches, too, but can also show rows in one table that have no match in the other table. Two kinds of outer joins are left and right joins. Most of the examples in this section use `LEFT JOIN`, which identifies rows in the left table that are not matched by the right table. `RIGHT JOIN` is the same except that the roles of the tables are reversed.

A `LEFT JOIN` works like this: You specify the columns to be used for matching rows in the two tables. When a row from the left table matches a row from the right table, the contents of the rows are selected as an output row. When a row in the left table has no match, it is still selected for output, but joined with a "fake" row from the right table that contains `NULL` in each column.

In other words, a `LEFT JOIN` forces the result set to contain a row for every row selected from the left table, whether or not there is a match for it in the right table. The left-table rows with no match can be identified by the fact that all columns from the right table are `NULL`. These result rows tell you which rows are missing from the right table. That is an interesting and important property, because this kind of problem comes up in many different contexts. Which customers have not been assigned an account representative? For which inventory items have no sales been recorded? Or, closer to home with our `sampdb` database: Which students have not taken a particular exam? Which students have no rows in the `absence` table (that is, which students have perfect attendance)?

Consider once again our two tables, `t1` and `t2`:

```
Table t1:     Table t2:
+----+----+   +----+----+
| i1 | c1 |   | i2 | c2 |
+----+----+   +----+----+
|  1 | a  |   |  2 | c  |
|  2 | b  |   |  3 | b  |
|  3 | c  |   |  4 | a  |
+----+----+   +----+----+
```

If we use an inner join to match these tables on `t1.i1` and `t2.i2`, we'll get output only for the values 2 and 3, because those are the values that appear in both tables:

```
mysql> SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ON t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  2 | b  |  2 | c  |
|  3 | c  |  3 | b  |
+----+----+----+----+
```

A left join produces output for every row in `t1`, whether or not `t2` matches it. To write a left join, name the tables with `LEFT JOIN` in between rather than `INNER JOIN`:

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
|  1 | a  | NULL | NULL |
|  2 | b  |    2 | c    |
|  3 | c  |    3 | b    |
+----+----+------+------+
```

Now there is an output row even for the `t1.i1` value of 1, which has no match in `t2`. All the columns in this row that correspond to `t2` columns have a value of `NULL`.

One thing to watch out for with `LEFT JOIN` is that unless right-table columns are defined as `NOT NULL`, you may get problematic rows in the result. For example, if the right table contains columns with `NULL` values, you won't be able to distinguish those `NULL` values from `NULL` values that identify unmatched rows.

As mentioned earlier, a `RIGHT JOIN` is like a `LEFT JOIN` with the roles of the tables reversed. These two statements are equivalent:

```
SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t2 RIGHT JOIN t1 ON t1.i1 = t2.i2;
```

The following discussion is phrased in terms of `LEFT JOIN`. To adjust it for `RIGHT JOIN`, reverse the table roles.

`LEFT JOIN` is especially useful when you want to find *only* those left table rows that are unmatched by the right table. Do this by adding a `WHERE` clause that selects only the rows that have `NULL` values in a right table column—in other words, the rows in one table that are missing from the other:

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
    -> WHERE t2.i2 IS NULL;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
|  1 | a  | NULL | NULL |
+----+----+------+------+
```

Normally, when you write a query like this, your real interest is in the unmatched values in the left table. The `NULL` columns from the right table are of no interest for display purposes, so you would omit them from the output column list:

```
mysql> SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
    -> WHERE t2.i2 IS NULL;
+----+----+
| i1 | c1 |
+----+----+
|  1 | a  |
+----+----+
```

Like `INNER JOIN`, a `LEFT JOIN` can be written using an `ON` clause or a `USING()` clause to specify the matching conditions. As with `INNER JOIN`, `ON` can be used whether or not the joined columns from each table have the same name, but `USING()` requires that they have the same names.

`NATURAL LEFT JOIN` is similar to `LEFT JOIN`; it performs a `LEFT JOIN`, matching all columns that have the same name in the left and right tables. (Thus, no `ON` or `USING` clause is given.)

As already mentioned, `LEFT JOIN` is useful for answering "Which values are missing?" questions. Let's apply this principle to the tables in the `sampdb` database and consider a more complex example than those shown earlier using `t1` and `t2`.

For the grade-keeping project, first mentioned in Chapter 1, "Getting Started with MySQL," we have a `student` table listing students, a `grade_event` table listing the grade events that have occurred, and a `score` table listing scores for each student for each grade event. However, if a student was ill on the day of some quiz or test, the `score` table wouldn't contain any score for the student for that event. A makeup quiz or test should be given in such cases, but how do we find these missing rows?

The problem is to determine which students have no score for a given grade event, and to do this for each grade event. That is, we want to find which combinations of student and grade event are not present in the `score` table. This "which values are not present" wording is a tip-off that we want a `LEFT JOIN`. The join isn't as simple as in the previous examples, though:

We aren't just looking for values that are not present in a single column, we're looking for a two-column combination. The combinations we want are all the student/event combinations. These are produced by joining the `student` table to the `grade_event` table:

```
FROM student INNER JOIN grade_event
```

Then we take the result of that join and perform a `LEFT JOIN` with the `score` table to find the matches for student ID/event ID pairs:

```
FROM student INNER JOIN grade_event
     LEFT JOIN score ON student.student_id = score.student.id
                     AND grade_event.event_id = score.event_id
```

Note that the `ON` clause causes the rows in the `score` table to be joined according to matches in different tables named earlier in the join. That's the key for solving this problem. The `LEFT JOIN` forces a row to be generated for each row produced by the join of the `student` and `grade_event` tables, even when there is no corresponding `score` table row. The result set rows for these missing score rows can be identified by the fact that the columns from the `score` table will all be `NULL`. We can identify these rows by adding a condition in the `WHERE` clause. Any column from the `score` table will do, but because we're looking for missing scores, it's probably conceptually clearest to test the `score` column:

```
WHERE score.score IS NULL
```

We can also sort the results using an `ORDER BY` clause. The two most logical orderings are by event per student and by student per event. I'll choose the first:

```
ORDER BY student.student_id, grade_event.event_id
```

Now all we need to do is name the columns we want to see in the output, and we're done. Here is the final statement:

```
SELECT
  student.name, student.student_id,
  grade_event.date, grade_event.event_id, grade_event.category
FROM
  student INNER JOIN grade_event
  LEFT JOIN score ON student.student_id = score.student_id
                  AND grade_event.event_id = score.event_id
WHERE
  score.score IS NULL
ORDER BY
  student.student_id, grade_event.event_id;
```

Running the query produces these results:

```
+-----------+------------+------------+----------+----------+
| name      | student_id | date       | event_id | category |
+-----------+------------+------------+----------+----------+
| Megan     |          1 | 2012-09-16 |        4 | Q        |
| Joseph    |          2 | 2012-09-03 |        1 | Q        |
```

```
| Katie     |          4 | 2012-09-23 |        5 | Q        |
| Devri     |         13 | 2012-09-03 |        1 | Q        |
| Devri     |         13 | 2012-10-01 |        6 | T        |
| Will      |         17 | 2012-09-16 |        4 | Q        |
| Avery     |         20 | 2012-09-06 |        2 | Q        |
| Gregory   |         23 | 2012-10-01 |        6 | T        |
| Sarah     |         24 | 2012-09-23 |        5 | Q        |
| Carter    |         27 | 2012-09-16 |        4 | Q        |
| Carter    |         27 | 2012-09-23 |        5 | Q        |
| Gabrielle |         29 | 2012-09-16 |        4 | Q        |
| Grace     |         30 | 2012-09-23 |        5 | Q        |
+-----------+------------+------------+----------+----------+
```

Here's a subtle point. The output displays the student IDs and the event IDs. The `student_id` column appears in both the `student` and `score` tables, so at first you might think that the output column list could name either `student.student_id` or `score.student_id`. That's not the case, because the entire basis for being able to find the rows we're interested in is that all the `score` table columns are returned by the `LEFT JOIN` as `NULL`. Selecting `score.student_id` would produce only a column of `NULL` values in the output. The same principle applies to deciding which `event_id` column to display. It appears in both the `grade_event` and `score` tables, but the query selects `grade_event.event_id` because the `score.event_id` values will always be `NULL`.

2.9 Performing Multiple-Table Retrievals with Subqueries

A subquery is a `SELECT` statement written within parentheses and nested inside another statement. Here's an example that looks up the IDs for grade event rows that correspond to tests (`'T'`) and uses them to select scores for those tests:

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

Subqueries can return different types of information:

- A scalar subquery returns a single value.
- A column subquery returns a single column of one or more values.
- A row subquery returns a single row of one or more values.
- A table subquery returns a table of one or more rows of one or more columns.

Subquery results can be tested in different ways:

- Scalar subquery results can be evaluated using relative comparison operators such as `=` or `<`.

- IN and NOT IN test whether a value is present in a set of values returned by a subquery.
- ALL, ANY, and SOME compare a value to the set of values returned by a subquery.
- EXISTS and NOT EXISTS test whether a subquery result is empty.

A scalar subquery is the most restrictive because it produces only a single value. But as a consequence, scalar subqueries can be used in the widest variety of contexts. They are applicable essentially anywhere that you can use a scalar operand, such as a term of an expression, as a function argument, or in the output column list. Column, row, and table subqueries that return more information cannot be used in contexts that require a single value.

Subqueries can be correlated or uncorrelated. This is a function of whether a subquery refers to and is dependent on values in the outer query.

You can use subqueries with statements other than SELECT. However, for statements that modify tables (DELETE, INSERT, REPLACE, UPDATE, LOAD DATA), MySQL enforces the restriction that the subquery cannot select from the table being modified.

In some cases, subqueries can be rewritten as joins. You might find subquery rewriting techniques useful to see whether the MySQL optimizer does a better job with a join than the equivalent subquery.

The following sections discuss the kinds of operations you can use to test subquery results, how to write correlated subqueries, and how to rewrite subqueries as joins.

2.9.1 Subqueries with Relative Comparison Operators

The =, <>, >, >=, <, and <= operators perform relative-value comparisons. When used with a scalar subquery, they find all rows in the outer query that stand in particular relationship to the value returned by the subquery. For example, to identify the scores for the quiz that took place on '2012-09-23', use a scalar subquery to determine the quiz event ID and then match score table rows against that ID in the outer SELECT:

```
SELECT * FROM score
WHERE event_id =
(SELECT event_id FROM grade_event
   WHERE date = '2012-09-23' AND category = 'Q');
```

With this form of statement, where the subquery is preceded by a value and a relative comparison operator, the subquery must produce a only single value. That is, it must be a scalar subquery; if it produces multiple values, the statement will fail. In some cases, it may be appropriate to satisfy the single-value requirement by limiting the subquery result with LIMIT 1.

Use of scalar subqueries with relative comparison operators is handy for solving problems for which you'd be tempted to use an aggregate function in a WHERE clause. For example, to determine which of the presidents in the president table was born first, you might try this statement:

```
SELECT * FROM president WHERE birth = MIN(birth);
```

That doesn't work because you can't use aggregates in WHERE clauses. (The WHERE clause determines which rows to select, but the value of MIN() isn't known until *after* the rows have already been selected.) However, you can use a subquery to produce the minimum birth date like this:

```
SELECT * FROM president
WHERE birth = (SELECT MIN(birth) FROM president);
```

Other aggregate functions can be used to solve similar problems. The following statement uses a subquery to select the above-average scores from a given grade event:

```
SELECT * FROM score WHERE event_id = 5
AND score > (SELECT AVG(score) FROM score WHERE event_id = 5);
```

If a subquery returns a single row, you can use a row constructor to compare a set of values (that is, a tuple) to the subquery result. This statement returns rows for presidents who were born in the same city and state as John Adams:

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) =
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Adams' AND first_name = 'John');
+-----------+------------+-----------+-------+
| last_name | first_name | city      | state |
+-----------+------------+-----------+-------+
| Adams     | John       | Braintree | MA    |
| Adams     | John Quincy | Braintree | MA   |
+-----------+------------+-----------+-------+
```

You can also use ROW(city, state) notation, which is equivalent to (city, state). Both act as row constructors.

2.9.2 IN and NOT IN Subqueries

The IN and NOT IN operators can be used when a subquery returns multiple rows to be evaluated in comparison to the outer query. They test whether a comparison value is present in a set of values. IN is true for rows in the outer query that match any row returned by the subquery. NOT IN is true for rows in the outer query that match no rows returned by the subquery. The following statements use IN and NOT IN to find those students who have absences listed in the absence table, and those who have perfect attendance (no absences):

```
mysql> SELECT * FROM student
    -> WHERE student_id IN (SELECT student_id FROM absence);
+-------+-----+------------+
| name  | sex | student_id |
+-------+-----+------------+
| Kyle  | M   |          3 |
| Abby  | F   |          5 |
```

```
| Peter | M   |         10 |
| Will  | M   |         17 |
| Avery | F   |         20 |
+-------+-----+------------+
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Katie     | F   |          4 |
| Nathan    | M   |          6 |
| Liesl     | F   |          7 |
...
```

IN and NOT IN also work for subqueries that return multiple columns. In other words, you can use them with table subqueries. In this case, use a row constructor to specify the comparison values to test against each column:

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) IN
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+-----------+-------------+-----------+-------+
```

IN and NOT IN actually are synonyms for = ANY and <> ALL, which are covered in the next section.

2.9.3 ALL, ANY, and SOME Subqueries

The ALL and ANY operators are used in conjunction with a relative comparison operator to test the result of a column subquery. They test whether the comparison value stands in particular relationship to all or some of the values returned by the subquery. For example, <= ALL is true if the comparison value is less than or equal to every value that the subquery returns, whereas <= ANY is true if the comparison value is less than or equal to any value that the subquery returns. SOME is a synonym for ANY.

This statement determines which president was born first by selecting the row with a birth date less than or equal to all the birth dates in the president table (only the earliest date satisfies this condition):

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ALL (SELECT birth FROM president);
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
+------------+------------+------------+
```

Less usefully, the following statement returns all rows because every date is less than or equal to at least one other date (itself):

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ANY (SELECT birth FROM president);
+------------+---------------+------------+
| last_name  | first_name    | birth      |
+------------+---------------+------------+
| Washington | George        | 1732-02-22 |
| Adams      | John          | 1735-10-30 |
| Jefferson  | Thomas        | 1743-04-13 |
| Madison    | James         | 1751-03-16 |
| Monroe     | James         | 1758-04-28 |
...
```

When ALL, ANY, or SOME are used with the = comparison operator, the subquery can be a table subquery. In this case, you test return rows using a row constructor to provide the comparison values.

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) = ANY
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+-----------+-------------+-----------+-------+
```

As mentioned in the previous section, IN and NOT IN are shorthand for = ANY and <> ALL. That is, IN means "equal to any of the rows returned by the subquery" and NOT IN means "unequal to all rows returned by the subquery."

2.9.4 EXISTS and NOT EXISTS Subqueries

The EXISTS and NOT EXISTS operators merely test whether a subquery returns any rows. If it does, EXISTS is true and NOT EXISTS is false. The following statements show some trivial examples of these subqueries. The first returns 0 if the absence table is empty, the second returns 1:

```
SELECT EXISTS (SELECT * FROM absence);
SELECT NOT EXISTS (SELECT * FROM absence);
```

`EXISTS` and `NOT EXISTS` actually are much more commonly used in correlated subqueries. For examples, see Section 2.9.5, "Correlated Subqueries."

With `EXISTS` and `NOT EXISTS`, the subquery uses `*` as the output column list. There's no need to name columns explicitly, because the subquery is assessed as true or false based on whether it returns any rows, not based on the particular values that the rows might contain. You can actually write pretty much anything for the subquery column selection list, but if you want to make it explicit that you're returning a true value when the subquery succeeds, you might write it as `SELECT 1` rather than `SELECT *`.

2.9.5 Correlated Subqueries

Subqueries can be uncorrelated or correlated:

- An uncorrelated subquery contains no references to values from the outer query, so it could be executed by itself as a separate statement. For example, the subquery in the following statement is uncorrelated because it refers only to the table `t1` and not to `t2`:

  ```
  SELECT j FROM t2 WHERE j IN (SELECT i FROM t1);
  ```

- A correlated subquery does contain references to values from the outer query, and thus is dependent on it. Due to this linkage, a correlated subquery cannot be executed by itself as a separate statement. For example, the subquery in the following statement is true for each value of column `j` in `t2` that matches a column `i` value in `t1`:

  ```
  SELECT j FROM t2 WHERE (SELECT i FROM t1 WHERE i = j);
  ```

Correlated subqueries commonly are used for `EXISTS` and `NOT EXISTS` subqueries, which are useful for finding rows in one table that match or don't match rows in another. Correlated subqueries work by passing values from the outer query to the subquery to see whether they match the conditions specified in the subquery. For this reason, it's necessary to qualify column names with table names if they are ambiguous (appear in more than one table).

The following `EXISTS` subquery identifies matches between the tables—that is, values that are present in both. The statement selects students who have at least one absence listed in the `absence` table:

```
SELECT student_id, name FROM student WHERE EXISTS
(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

`NOT EXISTS` identifies nonmatches—values in one table that are not present in the other. This statement selects students who have no absences:

```
SELECT student_id, name FROM student WHERE NOT EXISTS
(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

2.9.6 Subqueries in the FROM Clause

Subqueries can be used in the FROM clause to generate values. In this case, the result of the subquery acts like a table. A subquery in the FROM clause can participate in joins, its values can be tested in the WHERE clause, and so forth. With this type of subquery, you must provide a table alias to give the subquery result a name:

```
mysql> SELECT * FROM (SELECT 1, 2) AS t1 INNER JOIN (SELECT 3, 4) AS t2;
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
```

2.9.7 Rewriting Subqueries as Joins

It's often possible to rephrase a query that uses a subquery in terms of a join, and it's not a bad idea to examine queries that you might be inclined to write in terms of subqueries. A join is sometimes more efficient than a subquery, so if a SELECT written as a subquery takes a long time to execute, try writing it as a join to see whether it performs better. The following discussion shows how to do that.

2.9.7.1 Rewriting Subqueries That Select Matching Values

Here's an example statement containing a subquery; it selects scores from the score table only for tests (that is, it ignores quiz scores):

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

The same statement can be written without a subquery by converting it to a simple join:

```
SELECT score.* FROM score INNER JOIN grade_event
ON score.event_id = grade_event.event_id WHERE grade_event.category = 'T';
```

As another example, the following query selects scores for female students:

```
SELECT * from score
WHERE student_id IN (SELECT student_id FROM student WHERE sex = 'F');
```

This can be converted to a join as follows:

```
SELECT score.* FROM score INNER JOIN student
ON score.student_id = student.student_id WHERE student.sex = 'F';
```

There is a pattern here. The subquery statements follow this form:

```
SELECT * FROM table1
WHERE column1 IN (SELECT column2a FROM table2 WHERE column2b = value);
```

Such queries can be converted to a join using this form:

```
SELECT table1.* FROM table1 INNER JOIN table2
ON table1.column1 = table2.column2a WHERE table2.column2b = value;
```

In some cases, the subquery and the join might return different results. This occurs when *table2* contains multiple instances of *column2a*. The subquery form produces only one instance of each *column2a* value, but the join produces them all and its output includes duplicate rows. To suppress these duplicates, begin the join with `SELECT DISTINCT` rather than `SELECT`.

2.9.7.2 Rewriting Subqueries That Select Nonmatching (Missing) Values

Another common type of subquery statement searches for values in one table that are not present in another table. As we've seen before, the "which values are not present" type of problem is a clue that a `LEFT JOIN` may be helpful. Here's the statement with a subquery seen earlier that tests for students who are *not* listed in the `absence` table (it finds those students with perfect attendance):

```
SELECT * FROM student
WHERE student_id NOT IN (SELECT student_id FROM absence);
```

This query can be rewritten using a `LEFT JOIN` as follows:

```
SELECT student.*
FROM student LEFT JOIN absence ON student.student_id = absence.student_id
WHERE absence.student_id IS NULL;
```

In general terms, the subquery statement form is as follows:

```
SELECT * FROM table1
WHERE column1 NOT IN (SELECT column2 FROM table2);
```

A query having that form can be rewritten like this:

```
SELECT table1.*
FROM table1 LEFT JOIN table2 ON table1.column1 = table2.column2
WHERE table2.column2 IS NULL;
```

This assumes that *table2.column2* is defined as `NOT NULL`.

The subquery does have the advantage of being more intuitive than the `LEFT JOIN`. "Not in" is a concept that most people understand without difficulty, because it occurs outside the context of database programming. The same cannot be said for the concept of "left join," for which there is no such basis for natural understanding.

2.10 Performing Multiple-Table Retrievals with UNION

To create a result set that combines the results from several queries, use a `UNION` statement. For the examples in this section, assume that you have three tables, `t1`, `t2`, and `t3`, that look like this:

```
mysql> SELECT * FROM t1;
+------+-------+
| i    | c     |
+------+-------+
|    1 | red   |
|    2 | blue  |
|    3 | green |
+------+-------+
mysql> SELECT * FROM t2;
+------+------+
| j    | c    |
+------+------+
|   -1 | tan  |
|    1 | red  |
+------+------+
mysql> SELECT * FROM t3;
+------------+------+
| d          | k    |
+------------+------+
| 1904-01-01 |  100 |
| 2004-01-01 |  200 |
| 2004-01-01 |  200 |
+------------+------+
```

Tables `t1` and `t2` have integer and character columns, and `t3` has date and integer columns. To write a `UNION` statement that combines multiple retrievals, write multiple `SELECT` statements and put the keyword `UNION` between them. Each `SELECT` must retrieve the same number of columns. For example, to select the integer column from each table, do this:

```
mysql> SELECT i FROM t1 UNION SELECT j FROM t2 UNION SELECT k FROM t3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
|   -1 |
|  100 |
|  200 |
+------+
```

`UNION` has the following properties.

Column name and data types. The column names for the UNION result come from the names of the columns in the first SELECT. The second and subsequent SELECT statements in the UNION must select the same number of columns, but corresponding columns need not have the same names or data types. (Normally, you write a UNION such that corresponding columns do have the same types, but MySQL performs type conversion as necessary if they do not.) Column matching occurs by position rather than by name, which is why the following two statements return different results, even though they select the same values from the two tables:

```
mysql> SELECT i, c FROM t1 UNION SELECT k, d FROM t3;
+------+------------+
| i    | c          |
+------+------------+
|    1 | red        |
|    2 | blue       |
|    3 | green      |
|  100 | 1904-01-01 |
|  200 | 2004-01-01 |
+------+------------+
mysql> SELECT i, c FROM t1 UNION SELECT d, k FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
+------------+-------+
```

In each statement, the data type for each column of the result is determined from the selected values. In the first statement, strings and dates are selected for the second column. The result is a string column. In the second statement, integers and dates are selected for the first column, strings and integers for the second column. In both cases, the result is a string column.

Duplicate-row handling. By default, UNION eliminates duplicate rows from the result set:

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELECT * FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| -1         | tan   |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
+------------+-------+
```

t1 and t2 both have a row containing values of 1 and 'red', but only one such row appears in the output. Also, t3 has two rows containing '2004-01-01' and 200, one of which has been eliminated.

UNION DISTINCT is synonymous with UNION; both retain only distinct rows.

To preserve duplicates, change each UNION to UNION ALL:

```
mysql> SELECT * FROM t1 UNION ALL SELECT * FROM t2 UNION ALL SELECT * FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| -1         | tan   |
| 1          | red   |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
| 2004-01-01 | 200   |
+------------+-------+
```

If you mix UNION or UNION DISTINCT with UNION ALL, any distinct union operation takes precedence over any UNION ALL operations to its left.

ORDER BY and LIMIT handling. To sort a UNION result as a whole, place each SELECT within parentheses and add an ORDER BY clause following the last one. Because the UNION uses column names from the first SELECT, the ORDER BY should refer to those names, not the column names from the last SELECT:

```
mysql> (SELECT i, c FROM t1) UNION (SELECT k, d FROM t3)
    -> ORDER BY c;
+------+------------+
| i    | c          |
+------+------------+
|  100 | 1904-01-01 |
|  200 | 2004-01-01 |
|    2 | blue       |
|    3 | green      |
|    1 | red        |
+------+------------+
```

If a sort column is aliased, an ORDER BY at the end of the UNION must refer to the alias. Also, the ORDER BY cannot refer to table names. If you need to sort by a column specified as *tbl_name.col_name* in the first SELECT, alias the column and refer to the alias in the ORDER BY clause.

Similarly, to limit the number of rows returned by a UNION, add LIMIT to the end of the statement:

```
mysql> (SELECT * FROM t1) UNION (SELECT * FROM t2) UNION (SELECT * FROM t3)
    -> LIMIT 2;
+------+------+
| i    | c    |
+------+------+
| 1    | red  |
| 2    | blue |
+------+------+
```

`ORDER BY` and `LIMIT` also can be used within a parenthesized individual `SELECT` to apply only to that `SELECT`:

```
mysql> (SELECT * FROM t1 ORDER BY i LIMIT 2)
    -> UNION (SELECT * FROM t2 ORDER BY j LIMIT 1)
    -> UNION (SELECT * FROM t3 ORDER BY d LIMIT 2);
+------------+------+
| i          | c    |
+------------+------+
| 1          | red  |
| 2          | blue |
| -1         | tan  |
| 1904-01-01 | 100  |
| 2004-01-01 | 200  |
+------------+------+
```

`ORDER BY` within an individual `SELECT` is used only if `LIMIT` is also present, to determine which rows the `LIMIT` applies to. It does not affect the order in which rows appear in the final `UNION` result.

2.11 Multiple-Table Deletes and Updates

Sometimes it's useful to delete rows based on whether they match or don't match rows in another table. Similarly, it's often useful to update rows in one table using the contents of rows in another table. This section describes how to perform multiple-table `DELETE` and `UPDATE` operations. These types of statements draw heavily on the concepts used for joins, so be sure you're familiar with the material discussed earlier in Section 2.8, "Performing Multiple-Table Retrievals with Joins."

To perform a single-table `DELETE` or `UPDATE`, you refer only to the columns of one table and thus need not qualify the column names with the table name. For example, this statement deletes all rows in a table `t` that have `id` values greater than 100:

```
DELETE FROM t WHERE id > 100;
```

But what if you want to delete rows based not on properties inherent in the rows themselves, but rather on their relationship to rows in another table? Suppose that you want to delete from `t` those rows with `id` values that are present in or missing from another table `t2`?

To write a multiple-table `DELETE`, name all the tables in a `FROM` clause and specify the conditions used to match rows in the tables in the `WHERE` clause. The following statement deletes rows from table `t1` where there is a matching `id` value in table `t2`:

```
DELETE t1 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

Notice that if a column name appears in more than one of the tables, it is ambiguous and must be qualified with a table name.

The syntax also supports deleting rows from multiple tables at once. To delete rows from *both* tables where there are matching `id` values, name them both after the `DELETE` keyword:

```
DELETE t1, t2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

What if you want to delete nonmatching rows? A multiple-table `DELETE` can use any kind of join that you can write in a `SELECT`, so employ the same strategy that you'd use when writing a `SELECT` that identifies the nonmatching rows. That is, use a `LEFT JOIN` or `RIGHT JOIN`. For example, to identify rows in `t1` that have no match in `t2`, write a `SELECT` like this:

```
SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

The analogous `DELETE` statement to find and remove those rows from `t1` uses a `LEFT JOIN` as well:

```
DELETE t1 FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

MySQL supports a second multiple-table `DELETE` syntax. This syntax uses a `FROM` clause to list the tables from which rows are to be deleted and a `USING` clause to join the tables that determine which rows to delete. The preceding multiple-table `DELETE` statements can be rewritten using this syntax as follows:

```
DELETE FROM t1 USING t1 INNER JOIN t2 ON t1.id = t2.id;
DELETE FROM t1, t2 USING t1 INNER JOIN t2 ON t1.id = t2.id;
DELETE FROM t1 USING t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

The principles involved in writing multiple-table `UPDATE` statements are quite similar to those used for `DELETE`: Name all the tables that participate in the operation and qualify column references as necessary. Suppose that the quiz you gave on September 23, 2012 contained a question that everyone got wrong, and then you discover that the reason for this is that your answer key was incorrect. As a result, you want to add a point to everyone's score. With a multiple-table `UPDATE`, you can do this as follows:

```
UPDATE score, grade_event SET score.score = score.score + 1
WHERE score.event_id = grade_event.event_id
AND grade_event.date = '2012-09-23' AND grade_event.category = 'Q';
```

In this case, you could accomplish the same objective using a single-table update and a subquery:

```
UPDATE score SET score = score + 1
WHERE event_id = (SELECT event_id FROM grade_event
WHERE date = '2012-09-23' AND category = 'Q');
```

But other updates cannot be written using subqueries. For example, you might want to not only identify rows to update based on the contents of another table, but to copy column values from one table to another. The following statement copies `t1.a` to `t2.a` for rows that have a matching `id` column value:

```
UPDATE t1, t2 SET t2.a = t1.a WHERE t2.id = t1.id;
```

To perform multiple-table deletes or updates for InnoDB tables, you need not use the syntax just described. Instead, set up a foreign key relationship between tables that includes an `ON DELETE CASCADE` or `ON UPDATE CASCADE` constraint. For details, see Section 2.13, "Foreign Keys and Referential Integrity."

2.12 Performing Transactions

A transaction is a set of SQL statements that execute as a unit and can be canceled if necessary. Either all the statements execute successfully, or none of them have any effect. This is achieved through the use of commit and rollback capabilities. If all of the statements in the transaction succeed, you commit it to record their effects permanently in the database. If an error occurs during the transaction, you roll it back to cancel it. Any statements executed up to that point within the transaction are undone, leaving the database in the state it was in prior to the point at which the transaction began.

Commit and rollback provide the means to ensure that halfway-done operations don't make their way into your database and leave it in a partially updated (inconsistent) state. The canonical example involves a financial transfer where money from one account is placed into another account. Suppose that Bill writes a check to Bob for $100.00 and Bob cashes the check. Bill's account should be decremented by $100.00 and Bob's account incremented by the same amount:

```
UPDATE account SET balance = balance - 100 WHERE name = 'Bill';
UPDATE account SET balance = balance + 100 WHERE name = 'Bob';
```

If a crash occurs between the two statements, the operation is incomplete. Depending on which statement executes first, Bill is $100 short without Bob having been credited, or Bob is given $100 without Bill having been debited. Neither outcome is correct. If transactional capabilities are not available, you must figure out the state of ongoing operations at crash time by examining your logs manually to determine how to undo them or complete them. The rollback capabilities of transaction support enable you to handle this situation properly by undoing the effect of the statements that executed before the error occurred. (You may still have to determine which transactions weren't entered and re-issue them, but at least you don't have to worry about half-transactions making your database inconsistent.)

Another use for transactions is to make sure that the rows involved in an operation are not modified by other clients while you're working with them. MySQL automatically performs locking for single SQL statements to keep clients from interfering with each other, but this is not always sufficient to guarantee that a database operation achieves its intended result, because some operations are performed over the course of several statements. In this case, different clients might interfere with each other. A transaction groups statements into a single execution unit to prevent concurrency problems that could otherwise occur in a multiple-client environment.

Transactional systems typically are characterized as providing ACID properties. ACID is an acronym for Atomic, Consistent, Isolated, and Durable, referring to four properties transactions should have:

- **Atomicity:** The statements comprising a transaction form a logical unit. You can't have just some of them execute.
- **Consistency:** The database is consistent before and after the transaction executes. For example, if rows in one table cannot have an ID that is not listed in another table, a transaction that attempts to insert a row with an invalid ID will fail and roll back.
- **Isolation:** One transaction has no effect on another, so that transactions executed concurrently have the same effect as if done one after the other.
- **Durability:** When a transaction executes successfully to completion, its effects are recorded permanently in the database.

Transactional processing provides stronger guarantees about the outcome of database operations, but also requires more overhead in CPU cycles, memory, and disk space. MySQL offers storage engines that are transaction-safe (such as InnoDB), and that are not transaction-safe (such as MyISAM and MEMORY). Transactional properties are essential for some applications and not for others, and you can choose which ones make the most sense for your applications. Financial operations typically need transactions, and the guarantees of data integrity outweigh the cost of additional overhead. On the other hand, for an application that logs web page accesses to a database table, a loss of a few rows if the server host crashes might be tolerable. In this case, using a nontransactional storage engine avoids the overhead required for transactional processing.

2.12.1 Using Transactions to Ensure Safe Statement Execution

Use of transactions requires a transactional storage engine such as InnoDB. Engines such as MyISAM and MEMORY will not work. If you're not sure whether your MySQL server supports transactional storage engines, see Section 2.6.1.1, "Checking Which Storage Engines Are Available."

By default, MySQL runs in autocommit mode, which means that changes made by individual statements are committed to the database immediately to make them permanent. In effect, each statement is its own transaction implicitly. To perform transactions explicitly, disable autocommit mode and then tell MySQL when to commit or roll back changes.

One way to perform a transaction is to issue a `START TRANSACTION` (or `BEGIN`) statement to suspend autocommit mode, execute the statements that make up the transaction, and end the transaction with a `COMMIT` statement to make the changes permanent. If an error occurs during the transaction, cancel it by issuing a `ROLLBACK` statement instead to undo the changes.

START TRANSACTION suspends the current autocommit mode, so after the transaction has been committed or rolled back, the mode reverts to its state prior to the START TRANSACTION. If autocommit was enabled beforehand, ending the transaction puts you back in autocommit mode. If it was disabled, ending the current transaction causes you to begin the next one.

The following example illustrates this approach. First, create a table to use:

```
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE=InnoDB;
```

Next, initiate a transaction with START TRANSACTION, add a couple of rows to the table, commit the transaction, and then see what the table looks like:

```
mysql> START TRANSACTION;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

You can see that the rows have been recorded in the table. If you had started up a second instance of mysql and selected the contents of t after the inserts but before the commit, the rows would not show up. They would not become visible to the second mysql process until the COMMIT statement had been issued by the first one.

If an error occurs during a transaction, you can cancel it with ROLLBACK. Using the t table again, you can see this by issuing the following statements:

```
mysql> START TRANSACTION;
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062 (23000): Duplicate entry 'Wallace' for key 'name'
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

The second INSERT attempts to place a row into the table that duplicates an existing name value, but fails because name has a UNIQUE index. After issuing the ROLLBACK, the table has only the two rows that it contained prior to the failed transaction. In particular, the successful INSERT that was performed before the failed one has been undone and its effect is not recorded in the table.

Issuing a `START TRANSACTION` statement while a transaction is in process commits the current transaction implicitly before beginning a new one.

Another way to perform transactions is to manipulate the autocommit mode directly using `SET` statements:

```
SET autocommit = 0;
SET autocommit = 1;
```

Setting the `autocommit` variable to zero disables autocommit, The effects of any statements that follow become part of the current transaction, which you end by issuing a `COMMIT` or `ROLLBACK` statement to commit or cancel it. With this method, autocommit remains off until you turn it back on, so ending one transaction also begins the next one. You can also commit a transaction by re-enabling autocommit.

To see how this approach works, begin with the same table as for the previous examples:

```
mysql> DROP TABLE t;
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE=InnoDB;
```

Then disable autocommit mode, insert some rows, and commit the transaction:

```
mysql> SET autocommit = 0;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

At this point, the two rows have been committed to the table, but autocommit mode remains disabled. If you issue further statements, they become part of a new transaction, which may be committed or rolled back independently of the first transaction. To verify that autocommit is still off and that `ROLLBACK` will cancel uncommitted statements, issue the following statements:

```
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062 (23000): Duplicate entry 'Wallace' for key 'name'
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

To re-enable autocommit mode, use this statement:

```
mysql> SET autocommit = 1;
```

As just described, a transaction ends when you issue a `COMMIT` or `ROLLBACK` statement, or when you re-enable autocommit while it is disabled. Transactions also end under other circumstances. In addition to the `SET autocommit`, `START TRANSACTION`, `BEGIN`, `COMMIT`, and `ROLLBACK` statements that affect transactions explicitly, certain other statements do so implicitly because they cannot be part of a transaction. In general, these tend to be DDL (data definition language) statements that create, alter, or drop databases or objects in them, or statements that are lock-related. For example, if you issue any of the following statements while a transaction is in progress, the server commits the transaction first before executing the statement:

```
ALTER TABLE
CREATE INDEX
DROP DATABASE
DROP INDEX
DROP TABLE
LOCK TABLES
RENAME TABLE
SET autocommit = 1 (if not already set to 1)
TRUNCATE TABLE
UNLOCK TABLES (if tables currently are locked)
```

For a complete list of statements that cause implicit commits in your version of MySQL, see the MySQL Reference Manual.

A transaction also ends if a client's session ends or is broken before a commit occurs. In this case, the server automatically rolls back any transaction the client had in progress.

If a client program automatically reconnects after its session with the server is lost, the connection is reset to its default state of having autocommit enabled.

Transactions are useful in all kinds of situations. Suppose that you're working with the `score` table that is part of the grade-keeping project and you discover that the grades for two students have gotten mixed up and need to be switched. The incorrectly entered grades are as follows:

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          8 |        5 |    18 |
|          9 |        5 |    13 |
+------------+----------+-------+
```

To fix this, student 8 should be given a score of 13 and student 9 a score of 18. That can be done easily with two statements:

```
UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
```

However, it's necessary to ensure that both statements succeed as a unit. This is a problem to which transactional methods may be applied. To use `START TRANSACTION`, do this:

```
mysql> START TRANSACTION;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
```

To accomplish the same thing by manipulating the autocommit mode explicitly instead, do this:

```
mysql> SET autocommit = 0;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
mysql> SET autocommit = 1;
```

Either way, the result is that the scores are swapped properly:

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          8 |        5 |    13 |
|          9 |        5 |    18 |
+------------+----------+-------+
```

2.12.2 Using Transaction Savepoints

MySQL enables you to perform a partial rollback of a transaction. To do this, issue a `SAVEPOINT` statement within the transaction to set a named marker. To roll back to just that point in the transaction later, use a `ROLLBACK` statement that names the savepoint. The following statements illustrate how this works:

```
mysql> CREATE TABLE t (i INT) ENGINE=InnoDB;
mysql> START TRANSACTION;
mysql> INSERT INTO t VALUES(1);
mysql> SAVEPOINT my_savepoint;
mysql> INSERT INTO t VALUES(2);
mysql> ROLLBACK TO SAVEPOINT my_savepoint;
mysql> INSERT INTO t VALUES(3);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| i    |
+------+
|    1 |
|    3 |
+------+
```

After executing these statements, the first and third rows have been inserted, but the second one has been canceled by the partial rollback to the `my_savepoint` savepoint.

2.12.3 Transaction Isolation

Because MySQL is a multiple-user database system, different clients can attempt to use any given table at the same time. Storage engines such as MyISAM use table locking to keep clients from modifying a table at the same time, but this does not provide good concurrency performance when there are many updates. The InnoDB storage engine takes a different approach. It uses row-level locking for finer-grained control over table access by clients. One client can modify a row at the same time that another client reads or modifies a different row in the same table. If both clients want to modify a row at the same time, whichever of them acquires a lock on the row gets to modify it first. This provides better concurrency than table locking. However, there is the question of whether one client's transaction should be able to see the changes made by another client's transaction.

InnoDB implements transaction isolation levels to give clients control over what kind of changes made by other transactions they want to see. Different isolation levels permit or prevent problems that can occur when different transactions run simultaneously:

- Dirty reads. A dirty read occurs when a change made by one transaction can be seen by other transactions before the transaction has been committed. Another transaction thus might think the row has been changed, even though that will not really be true if the transaction that changed the row later is rolled back.
- Nonrepeatable reads. A nonrepeatable read refers to failure by a transaction to get the same result for a given `SELECT` statement each time it executes it. This might happen if one transaction performs a `SELECT` twice but another transaction changes some of the rows in between the two executions.
- Phantom rows. A phantom is a row that becomes visible to a transaction when it was not previously. Suppose that a transaction performs a `SELECT` and then another transaction inserts a row. If the first transaction runs the same `SELECT` again and sees the new row, that is a phantom.

To deal with these problems, InnoDB supports four transaction isolation levels. These levels determine which modifications made by one transaction can be seen by other transactions that execute at the same time:

- `READ UNCOMMITTED`: A transaction can see row modifications made by other transactions even before they have been committed.
- `READ COMMITTED`: A transaction can see row modifications made by other transactions only if they have been committed.
- `REPEATABLE READ`: If a transaction performs a given `SELECT` twice, the result is repeatable. That is, it gets the same result each time, even if other transactions have changed or inserted rows in the meantime.

- `SERIALIZABLE`: This isolation level is similar to `REPEATABLE READ` but isolates transactions more completely: Rows examined by one transaction cannot be modified by other transactions until the first transaction completes. This enables one transaction to read rows and at the same time prevent them from being modified by other transactions until it is done with them.

Table 2.4 shows for each isolation level whether it permits dirty reads, nonrepeatable reads, or phantom rows. The table is InnoDB-specific in that `REPEATABLE READ` does not permit phantom rows to occur. Some database systems do permit phantoms at the `REPEATABLE READ` isolation level.

Table 2.4 **Problems Permitted by Isolation Levels**

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantom Rows
`READ UNCOMMITTED`	Yes	Yes	Yes
`READ COMMITTED`	No	Yes	Yes
`REPEATABLE READ`	No	No	No
`SERIALIZABLE`	No	No	No

The default InnoDB isolation level is `REPEATABLE READ`. This can be changed at server startup with the `--transaction-isolation` option, or at runtime with the `SET TRANSACTION` statement. The statement has three forms:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL level;
SET SESSION TRANSACTION ISOLATION LEVEL level;
SET TRANSACTION ISOLATION LEVEL level;
```

A client that has the `SUPER` privilege can use `SET TRANSACTION` to change the global isolation level, which then applies to any clients that connect thereafter. In addition, any client can change its own transaction isolation level, either for all subsequent transactions within its session with the server (if `SESSION` is specified) or for its next transaction only (if `SESSION` is omitted). No special privileges are required for the client-specific levels.

> ### Can You Mix Transactional and Nontransactional Tables?
> It is possible to use both transactional and nontransactional tables during the course of a transaction, but the result might not be what you expect. Statements for nontransactional tables always take effect immediately, even when `autocommit` is disabled. In effect, nontransactional tables are always in autocommit mode and each statement commits immediately. As a result, if you change a nontransactional table within a transaction and then attempt a rollback, the nontransactional table changes cannot be undone.

2.13 Foreign Keys and Referential Integrity

A foreign key relationship enables you to declare that an index in one table is related to an index in another. It also enables you to place constraints on what may be done to the tables in the relationship. The database enforces the rules of this relationship to maintain referential integrity. For example, the `score` table in the `sampdb` sample database contains a `student_id` column, which we use to relate score rows to students in the `student` table. When we created these tables in Chapter 1, "Getting Started with MySQL," we set up some explicit relationships between them. For example, we declared `score.student_id` to be a foreign key for the `student.student_id` column. That prevents a row from being entered into the `score` table unless its `student_id` value exists in the `student` table. In other words, the foreign key prevents entry of scores for nonexistent students.

Foreign keys are not useful just for row entry, but for deletes and updates as well. For example, we could set up a constraint such that if a student is deleted from the `student` table, all corresponding rows for the student in the `score` table are deleted automatically as well. This is called "cascaded delete" because the effect of the delete cascades from one table to another. Cascaded update is possible as well. For example, with cascaded update, changing a student's `student_id` value in the `student` table also changes the value in the student's corresponding `score` table rows.

Foreign keys maintain the consistency of your data, and they provide a certain measure of convenience. Without foreign keys, you are responsible for keeping track of inter-table dependencies and maintaining their consistency from within your applications. In some cases, doing this might not be much more work than issuing a few extra `DELETE` statements to make sure that when you delete a row from one table, you also delete the corresponding rows in any related tables. But it *is* extra work, and if the database engine will perform consistency checks for you, why not let it? Automatic checking capability is especially useful if your tables have particularly complex relationships. You likely will not want to be responsible for implementing these dependencies in your applications.

In MySQL, the InnoDB storage engine provides foreign key support. This section describes how to set up InnoDB tables to define foreign keys, and how foreign keys affect the way you use tables. First, it's necessary to define some terms:

- The parent is the table that contains the original key values.
- The child is the related table that refers to key values in the parent.

Parent table key values are used to associate the two tables. Specifically, an index in the child table refers to an index in the parent. The child index values must match those in the parent or else be set to `NULL` to indicate that there is no associated parent table row. The index in the child table is known as the "foreign key"—that is, the key that is foreign (external) to the parent table but contains values that point to the parent. A foreign key relationship can be set up to reject `NULL` values, in which case all foreign key values must match a value in the parent table.

InnoDB enforces these rules to guarantee that the foreign key relationship stays intact with no mismatches. This is called "referential integrity."

The following syntax shows how to define a foreign key in a child table:

```
[CONSTRAINT constraint_name]
FOREIGN KEY [fk_name] (index_columns)
  REFERENCES tbl_name (index_columns)
  [ON DELETE action]
  [ON UPDATE action]
  [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
```

Although all parts of this syntax are parsed, InnoDB does not implement the semantics for all the clauses: The `MATCH` clause is not supported and is ignored if you specify it. Also, some *`action`* values are recognized but have no effect. (For storage engines other than InnoDB, the entire `FOREIGN KEY` definition is parsed but ignored.)

InnoDB pays attention to the following parts of the definition:

- The `CONSTRAINT` clause, if given, supplies a name for the foreign key constraint. If you omit it, InnoDB creates a name.
- `FOREIGN KEY` indicates the indexed columns in the child table that must match index values in the parent table. *`fk_name`* is the foreign key ID. If given, it is ignored unless InnoDB automatically creates an index for the foreign key; in that case, *`fk_name`* becomes the index name.
- `REFERENCES` names the parent table and the index columns in that table to which the foreign key in the child table refers. The *`index_columns`* part of the `REFERENCES` clause must have the same number of columns as the *`index_columns`* that follows the `FOREIGN KEY` keywords.
- `ON DELETE` enables you to specify what happens to the child table when parent table rows are deleted. The default if no `ON DELETE` clause is present is to reject any attempt to delete rows in the parent table that have child rows pointing to them. To specify an *`action`* value explicitly, use one of the following clauses:
    - `ON DELETE NO ACTION` and `ON DELETE RESTRICT` are the same as omitting the `ON DELETE` clause. Some database systems have deferred checks, and `NO ACTION` is a deferred check. For InnoDB, foreign key constraints are checked immediately, so `NO ACTION` and `RESTRICT` are the same.
    - `ON DELETE CASCADE` causes matching child rows to be deleted when the corresponding parent row is deleted. In essence, the effect of the delete is cascaded from the parent to the child. This enables you to perform multiple-table deletes by deleting rows only from the parent table and letting InnoDB delete the corresponding rows from the child table.
    - `ON DELETE SET NULL` causes index columns in matching child rows to be set to `NULL` when the parent row is deleted. If you use this option, all the indexed child table columns named in the foreign key definition must be defined to permit `NULL` values. (One implication of using this action is that you cannot define the foreign key to be a `PRIMARY KEY`; primary keys do not permit `NULL` values.)
    - `ON DELETE SET DEFAULT` is recognized but unimplemented and InnoDB issues an error.

- `ON UPDATE` enables you to specify what happens to the child table when parent table rows are updated. The default if no `ON UPDATE` clause is present is to reject any inserts or updates in the child table that result in foreign key values that don't have any match in the parent table index, and to prevent updates to parent table index values to which child rows point. The possible *`action`* values are the same as for `ON DELETE` and have similar effects.

To set up a foreign key relationship, follow these guidelines:

- The child table must have an index where the foreign key columns are listed as its first columns. The parent table must also have an index in which the columns in the `REFERENCES` clause are listed as its first columns. (In other words, the key columns must be indexed in the tables on both ends of the foreign key relationship.) You must create the parent table index explicitly before defining the foreign key relationship. InnoDB automatically creates an index on foreign key columns (the referencing columns) in the child table if the `CREATE TABLE` statement does not include such an index. This makes it easier to write the `CREATE TABLE` statement in some cases. However, an automatically created index will be a nonunique index and will include only the foreign key columns. You should define the index in the child table explicitly if you want it to be a `PRIMARY KEY` or `UNIQUE` index, or if it should include other columns in addition to those in the foreign key.
- Corresponding columns in the parent and child indexes must have compatible types. For example, you cannot match an `INT` column with a `CHAR` column. Corresponding character columns must be the same length. Corresponding integer columns must have the same size and must both be signed or both `UNSIGNED`.
- You cannot index prefixes of string columns in foreign key relationships. (That is, for string columns, you must index the entire column, not just a leading prefix of it.)

In Chapter 1, "Getting Started with MySQL," we created tables for the grade-keeping project that have simple foreign key relationships. Now let's work through an example that is more complex. Begin by creating tables named `parent` and `child`, such that the `child` table contains a foreign key that references the `par_id` column in the `parent` table:

```
CREATE TABLE parent
(
  par_id    INT NOT NULL,
  PRIMARY KEY (par_id)
) ENGINE = INNODB;

CREATE TABLE child
(
  par_id    INT NOT NULL,
  child_id  INT NOT NULL,
  PRIMARY KEY (par_id, child_id),
  FOREIGN KEY (par_id) REFERENCES parent (par_id)
```

```
    ON DELETE CASCADE
    ON UPDATE CASCADE
) ENGINE = INNODB;
```

The foreign key in this case uses `ON DELETE CASCADE` to specify that when a row is deleted from the `parent` table, MySQL also should remove `child` rows with a matching `par_id` value automatically. `ON UPDATE CASCADE` indicates that if a parent row `par_id` value is changed, MySQL also should change any matching `par_id` values in the `child` table to the new value.

Now insert a few rows into the `parent` table, and then add some rows to the `child` table that have related key values:

```
mysql> INSERT INTO parent (par_id) VALUES(1),(2),(3);
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
```

These statements result in the following table contents, where each `par_id` value in the `child` table matches a `par_id` value in the `parent` table:

```
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      1 |
|      2 |
|      3 |
+--------+
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      1 |        1 |
|      1 |        2 |
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

To verify that InnoDB enforces the key relationship for insertion, try adding a row to the `child` table that has a `par_id` value not found in the `parent` table:

```
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`child`, CONSTRAINT `child_ibfk_1` FOREIGN
KEY (`par_id`) REFERENCES `parent` (`par_id`) ON DELETE CASCADE
ON UPDATE CASCADE)
```

To test cascaded delete, see what happens when you delete a parent row:

```
mysql> DELETE FROM parent WHERE par_id = 1;
```

MySQL deletes the row from the parent table:

```
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      2 |
|      3 |
+--------+
```

In addition, it cascades the effect of the DELETE statement to the child table:

```
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

To test cascaded update, see what happens when you update a parent row:

```
mysql> UPDATE parent SET par_id = 100 WHERE par_id =2;
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      3 |
|    100 |
+--------+
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      3 |        1 |
|    100 |        1 |
|    100 |        2 |
|    100 |        3 |
+--------+----------+
```

The preceding example shows how to arrange for deletes or updates of a parent row to cause cascaded deletes or updates of any corresponding child rows. The ON DELETE and ON UPDATE clauses permit other actions. For example, one possibility is to let the child rows remain in the table but have their foreign key columns set to NULL. To do this, it's necessary to make several changes to the definition of the child table:

- Use ON DELETE SET NULL rather than ON DELETE CASCADE. This tells InnoDB to set the foreign key column (par_id) to NULL instead of deleting the rows.
- Use ON UPDATE SET NULL rather than ON UPDATE CASCADE. This tells InnoDB to set the foreign key column (par_id) to NULL when matching parent rows are updated.
- The original definition of child defines par_id as NOT NULL. That won't work with ON DELETE SET NULL or ON UPDATE SET NULL, so the column definition must be changed to permit NULL.
- The original definition of child also defines par_id to be part of a PRIMARY KEY. However, a PRIMARY KEY cannot contain NULL values. Changing par_id to permit NULL therefore also requires that the PRIMARY KEY be changed to a UNIQUE index. UNIQUE indexes enforce uniqueness except for NULL values, which can occur multiple times in the index.

To see the effect of these changes, re-create the parent table using the original definition and load the same initial rows into it. Then create the child table using the new definition shown here:

```
CREATE TABLE child
(
  par_id    INT NULL,
  child_id  INT NOT NULL,
  UNIQUE (par_id, child_id),
  FOREIGN KEY (par_id) REFERENCES parent (par_id)
    ON DELETE SET NULL
    ON UPDATE SET NULL
) ENGINE = INNODB;
```

With respect to inserting new rows, the child table behaves similarly to the original definition. That is, it permits insertion of rows with par_id values found in the parent table, but prohibits entry of values that aren't listed there:

```
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`child`, CONSTRAINT `child_ibfk_1` FOREIGN
KEY (`par_id`) REFERENCES `parent` (`par_id`) ON DELETE SET NULL
ON UPDATE SET NULL)
```

There is one difference with respect to inserting rows. Because the par_id column now is defined as NULL, you can explicitly insert rows into the child table that contain NULL and no error occurs. A difference in behavior also occurs when you delete a parent row. Try removing a parent row and then check the contents of the child table to see what happens:

```
mysql> DELETE FROM parent WHERE par_id = 1;
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|   NULL |        1 |
|   NULL |        2 |
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

In this case, the child rows that had `1` in the `par_id` column are not deleted. Instead, the `par_id` column is set to `NULL`, as specified by the `ON DELETE SET NULL` constraint.

Updating a parent row has a similar effect:

```
mysql> UPDATE parent SET par_id = 100 WHERE par_id = 2;
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|   NULL |        1 |
|   NULL |        1 |
|   NULL |        2 |
|   NULL |        2 |
|   NULL |        3 |
|      3 |        1 |
+--------+----------+
```

To see what foreign key relationships an InnoDB table has, use the `SHOW CREATE TABLE` statement.

If an error occurs when you attempt to create a table that has a foreign key, use the `SHOW ENGINE INNODB STATUS` statement to get the full error message.

2.14 Using FULLTEXT Searches

MySQL is capable of performing full-text searches, which enables you to look for words or phrases without using pattern-matching operations. There are three kinds of full-text search:

- Natural language searching (the default). MySQL parses the search string into words and searches for rows containing these words.
- Boolean mode searching. Words in the search string can include modifier characters that indicate specific requirements, such as that a given word should be present or absent in matching rows, or that rows must contain an exact phrase.

- Query expansion searching. This kind of search occurs in two phases. The first phase is a natural language search. Then a second search is done using the original search string concatenated with the most highly relevant matching rows from the first search. This expands the search on the basis of the assumption that words related to the original search string will match relevant rows that the original string did not.

Full-text search capability is enabled for a given table by creating a special kind of index and has the following characteristics:

- Full-text searches are based on `FULLTEXT` indexes. In MySQL 5.5, these can be created only for MyISAM tables. MySQL 5.6 introduces full-text support for InnoDB, but we'll stick with MyISAM here because you might not have 5.6. Only `CHAR`, `VARCHAR`, and `TEXT` columns can be included in a `FULLTEXT` index.
- Common words are ignored for `FULLTEXT` searches, where "common" means "present in at least half the rows." It's especially important to remember this when you're setting up a test table to experiment with the `FULLTEXT` capability. Be sure to insert at least three rows into your test table. If the table has just one or two rows, every word in it will occur at least 50% of the time and you'll never get any results!
- There is a built-in list of common words such as "the," "after," and "other" that are called "stopwords" and that are always ignored.
- Words that are too short are ignored. By default, "too short" is defined as fewer than four characters, but you can reconfigure the server to set the minimum length to a different value. (See Section 2.14.4, "Configuring the `FULLTEXT` Search Engine".)
- Words are defined as sequences of characters that include letters, digits, apostrophes, and underscores. This means that a string like "full-blooded" is considered to contain two words, "full" and "blooded." Normally, a full-text search matches whole words, not partial words, and the `FULLTEXT` engine considers a row to match a search string if it includes any of the words in the search string. If you use a boolean full-text search, you can impose the additional constraint that all the words must be present (either in any order, or, to perform a phrase search, in exactly the order listed in the search string). With a boolean search, it's also possible to match rows that do *not* include certain words, or to add a wildcard modifier to match all words that begin with a given prefix.
- A `FULLTEXT` index can be created for a single column or multiple columns. If it spans multiple columns, searches based on the index look through all the columns simultaneously. The flip side of this is that when you perform a search, you must specify a column list that corresponds exactly to the set of columns that matches some `FULLTEXT` index. For example, if you want to search `col1` sometimes, `col2` sometimes, and both `col1` and `col2` sometimes, you must create three indexes: one for each of the columns separately, and one that includes both columns.

The following examples show how to use full-text searching by creating `FULLTEXT` indexes and then performing queries on them using the `MATCH` operator. A script to create the table and some sample data to load into it are available in the `full-text` directory of the `sampdb` distribution.

Create a FULLTEXT index much the same way as other indexes: Define it with CREATE TABLE when creating the table initially, or add it afterward with ALTER TABLE or CREATE INDEX. Because FULLTEXT indexes require you to use MyISAM tables, you can take advantage of one of the properties of the MyISAM storage engine if you're creating a new table to use for FULLTEXT searches: Table loading proceeds more quickly if you populate the table first and then add the indexes afterward, rather than loading data into an already indexed table. Suppose that you have a data file named apothegm.txt containing famous sayings and the people to whom they're attributed:

```
Aeschylus               Time as he grows old teaches many lessons
Alexander Graham Bell   Mr. Watson, come here. I want you!
Benjamin Franklin       It is hard for an empty bag to stand upright
Benjamin Franklin       Little strokes fell great oaks
Benjamin Franklin       Remember that time is money
Miguel de Cervantes     Bell, book, and candle
Proverbs 15:1           A soft answer turneth away wrath
Theodore Roosevelt      Speak softly and carry a big stick
William Shakespeare     But, soft! what light through yonder window breaks?
Robert Burton           I light my candle from their torches.
```

If you want to search by phrase and attribution separately or together, you need to index each column separately, and also create an index that includes both columns. You can create, populate, and index a table named apothegm as follows:

```
CREATE TABLE apothegm (attribution VARCHAR(40), phrase TEXT) ENGINE=MyISAM;
LOAD DATA LOCAL INFILE 'apothegm.txt' INTO TABLE apothegm;
ALTER TABLE apothegm
  ADD FULLTEXT (phrase),
  ADD FULLTEXT (attribution),
  ADD FULLTEXT (phrase, attribution);
```

2.14.1 Natural Language FULLTEXT Searches

After setting up the table, perform natural language full-text searches on it using MATCH to name the column or columns to search and AGAINST() to specify the search string. For example:

```
mysql> SELECT * FROM apothegm WHERE MATCH(attribution) AGAINST('roosevelt');
+--------------------+------------------------------------+
| attribution        | phrase                             |
+--------------------+------------------------------------+
| Theodore Roosevelt | Speak softly and carry a big stick |
+--------------------+------------------------------------+
mysql> SELECT * FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+-------------------+-------------------------------------------+
| attribution       | phrase                                    |
+-------------------+-------------------------------------------+
```

```
| Benjamin Franklin | Remember that time is money               |
| Aeschylus         | Time as he grows old teaches many lessons |
+-------------------+-------------------------------------------+
mysql> SELECT * FROM apothegm WHERE MATCH(attribution, phrase)
    -> AGAINST('bell');
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes   | Bell, book, and candle            |
+-----------------------+-----------------------------------+
```

In the last example, note how the query finds rows that contain the search word in different columns, which demonstrates the FULLTEXT capability of searching multiple columns at once. Also note that the order of the columns as named in the query is attribution, phrase. That differs from the order in which they were named when the index was created (phrase, attribution), which illustrates that order does not matter. What matters is that there must be some FULLTEXT index that consists of exactly the columns named.

To see only how many rows a search matches, use COUNT(*):

```
mysql> SELECT COUNT(*) FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

Output rows for natural language FULLTEXT searches are ordered by decreasing relevance when you use a MATCH expression in the WHERE clause. Relevance values are nonnegative floating point values, with zero indicating "no relevance." To see these values, use a MATCH expression in the output column list:

```
mysql> SELECT phrase, MATCH(phrase) AGAINST('time') AS relevance
    -> FROM apothegm;
+-----------------------------------------------------+--------------------+
| phrase                                              | relevance          |
+-----------------------------------------------------+--------------------+
| Time as he grows old teaches many lessons           | 1.3253291845321655 |
| Mr. Watson, come here. I want you!                  |                  0 |
| It is hard for an empty bag to stand upright        |                  0 |
| Little strokes fell great oaks                      |                  0 |
| Remember that time is money                         |  1.340062141418457 |
| Bell, book, and candle                              |                  0 |
| A soft answer turneth away wrath                    |                  0 |
| Speak softly and carry a big stick                  |                  0 |
| But, soft! what light through yonder window breaks? |                  0 |
| I light my candle from their torches.               |                  0 |
+-----------------------------------------------------+--------------------+
```

A natural language search finds rows that contain any of the search words, so a query such as the following returns rows that contain either “hard” or “soft”:

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
    -> AGAINST('hard soft');
+--------------------+-------------------------------------------------+
| attribution        | phrase                                          |
+--------------------+-------------------------------------------------+
| Benjamin Franklin  | It is hard for an empty bag to stand upright    |
| Proverbs 15:1      | A soft answer turneth away wrath                |
| William Shakespeare | But, soft! what light through yonder window breaks? |
+--------------------+-------------------------------------------------+
```

Natural language mode is the default full-text search mode. To specify this mode explicitly, add `IN NATURAL LANGUAGE MODE` after the search string. The following statement performs the same search as the preceding example:

```
SELECT * FROM apothegm WHERE MATCH(phrase)
AGAINST('hard soft' IN NATURAL LANGUAGE MODE);
```

2.14.2 Boolean Mode FULLTEXT Searches

Greater control over multiple-word matching can be obtained by using boolean mode `FULLTEXT` searches. This type of search is performed by adding `IN BOOLEAN MODE` after the search string in the `AGAINST()` function. Boolean searches have the following characteristics:

- The 50% rule is ignored. Searches find words even if they occur in more than half of the rows.
- No sorting by relevance occurs.
- A search can require all words in a phrase to be present in a particular order. To match a phrase, specify it within double quotes. Matches occur for rows that contain the same words together in the same order as listed in the phrase:

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('"bell book and candle"' IN BOOLEAN MODE);
+--------------------+-----------------------+
| attribution        | phrase                |
+--------------------+-----------------------+
| Miguel de Cervantes | Bell, book, and candle |
+--------------------+-----------------------+
```

- It’s possible to perform a boolean mode full-text search on columns that are not part of a `FULLTEXT` index, although this is much slower than using indexed columns.

For boolean searches, modifiers may be applied to words in the search string. A leading plus or minus sign requires a word to be present or not present in matching rows. For example, a search string of `'bell'` matches rows that contain “bell,” but a search string of `'+bell`

`-candle'` in boolean mode matches only rows that contain "bell" and do not contain "candle."

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell');
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes   | Bell, book, and candle            |
+-----------------------+-----------------------------------+
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('+bell -candle' IN BOOLEAN MODE);
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
+-----------------------+-----------------------------------+
```

A trailing asterisk acts as a wildcard so that any row containing words beginning with the search word match. For example `'soft*'` matches "soft," "softly," "softness," and so forth:

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
    -> AGAINST('soft*' IN BOOLEAN MODE);
+--------------------+-------------------------------------------------+
| attribution        | phrase                                          |
+--------------------+-------------------------------------------------+
| Proverbs 15:1      | A soft answer turneth away wrath                |
| William Shakespeare | But, soft! what light through yonder window breaks?|
| Theodore Roosevelt  | Speak softly and carry a big stick              |
+--------------------+-------------------------------------------------+
```

However, the wildcard feature cannot be used to match words shorter than the minimum index word length.

The entry for `MATCH` in Appendix C, "Operator and Function Reference," lists the full set of boolean mode modifiers.

Stopwords are ignored just as for natural language searches, even if marked as required. A search for `'+Alexander +the +great'` finds rows containing "Alexander" and "great," but ignores "the" as a stopword.

2.14.3 Query Expansion FULLTEXT Searches

A full-text search with query expansion performs a two-phase search. The initial search is like a regular natural language search. Then the most highly relevant rows from this search are used for the second phase. The words in these rows are used along with the original search terms to perform a second search. Because the set of search terms is larger, the result generally includes rows that are not found in the first phase but are related to them.

To perform this kind of search, add `WITH QUERY EXPANSION` following the search terms. The following example provides an illustration. The first query shows a natural language search. The second query shows a query expansion search. Its result includes an extra row that contains none of the original search terms. This row is found because it contains the word "candle" that is present in one of the rows found by the natural language search.

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell book');
+-----------------------+------------------------------------+
| attribution           | phrase                             |
+-----------------------+------------------------------------+
| Miguel de Cervantes   | Bell, book, and candle             |
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
+-----------------------+------------------------------------+
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell book' WITH QUERY EXPANSION);
+-----------------------+-----------------------------------------+
| attribution           | phrase                                  |
+-----------------------+-----------------------------------------+
| Miguel de Cervantes   | Bell, book, and candle                  |
| Alexander Graham Bell | Mr. Watson, come here. I want you!      |
| Robert Burton         | I light my candle from their torches.   |
+-----------------------+-----------------------------------------+
```

2.14.4 Configuring the FULLTEXT Search Engine

Several full-text parameters are configurable and can be modified by setting system variables. The `ft_min_word_len` and `ft_max_word_len` variables determine the shortest and longest words to index in `FULLTEXT` indexes. Words with lengths outside the range defined by these two variables are ignored when `FULLTEXT` indexes are built. The default minimum and maximum values are 4 and 84.

Suppose that you want to change the minimum word length from 4 to 3. Do so like this:

1. Start the server with the `ft_min_word_len` variable set to 3. To ensure that this happens whenever the server starts, it's best to place the setting in an option file such as `/etc/my.cnf`:

   ```
   [mysqld]
   ft_min_word_len=3
   ```

2. For any existing tables that already have `FULLTEXT` indexes, you must rebuild those indexes. You can drop and add the indexes, but it's easier and sufficient to perform a quick repair operation:

   ```
   REPAIR TABLE tbl_name QUICK;
   ```

3. Any new `FULLTEXT` indexes that you create after changing the parameter will use the new value automatically.

For more information on setting system variables, see Appendix D, "System, Status, and User Variable Reference." For details on using option files, see Appendix F, "MySQL Program Reference."

Note

If you use `myisamchk` to rebuild indexes for a table that contains any `FULLTEXT` indexes, see the `FULLTEXT`-related notes in the `myisamchk` description in Appendix F, "MySQL Program Reference."

This page intentionally left blank

Index

Symbols

A

B

C

E

G

I

J

N

O

P

Q

R

S

T

U

V