Ben Forta

Updated for **MYSQL 8**

MySQL®

CRASH COURSE

SECOND EDITION

GET UP AND RUNNING WITH MYSQL

MASTER MYSQL WORKBENCH

LEARN HOW TO RETRIEVE, SORT, AND FILTER DATA

TAKE ADVANTAGE OF REGULAR EXPRESSION FILTERING AND FULL TEXT SEARCHING

DISCOVER POWERFUL MYSQL FEATURES, INCLUDEING STORED PROCEDURES AND TRIGGERS

USE VIEWS AND CURSORS

MANAGE TRANSACTIONAL PROCESSING

CREATE USER ACCOUNTS AND MANAGE SECURITY VIA ACCESS CONTROL

MySQL Crash Course

This page intentionally left blank

MySQL Crash Course

Second Edition

Ben Forta

Addison-Wesley

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2023943569

Hoboken, New Jersey

ISBN-13: 978-0-13-822302-1
ISBN-10: 0-13-822302-5

$PrintCode

Vice President, IT Professional
Mark Taub

Acquisitions Editor
Kim Spenceley

Development Editor
Chris Zahn

Managing Editor
Sandra Schroeder

Senior Project Editor
Tonya Simpson

Copy Editor
Kitty Wilson

Indexer
Timothy Wright

Proofreader
Jennifer Hinchliffe

Editorial Assistant
Cindy Teeters

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where

- Everyone has an equitable and lifelong opportunity to succeed through learning
- Our educational products and services are inclusive and represent the rich diversity of learners
- Our educational content accurately reflects the histories and experiences of the learners we serve
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview)

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

Contents

This page intentionally left blank

Acknowledgments

Thanks to the team at Pearson for all these years of support, dedication, and encouragement. Over the past two and a half decades, we've created 40+ books together, but our little *Sams Teach Yourself SQL in 10 Minutes* series remains my favorite by far. Thank you for trusting me with the creative freedom to evolve it as I have seen fit.

Speaking of *Sams Teach Yourself SQL in 10 Minutes*, that title covers MySQL (as it does all major DBMSs), but it cannot provide in-depth lessons on features that are truly unique to MySQL. This spinoff book was written in response to numerous requests from readers for greater MySQL-specific coverage. Thanks for the nudge. I hope this book lives up to your expectations.

Thanks to the many thousands of readers who provided feedback on prior editions of these books. Fortunately, most of it was positive; all of it was appreciated. The enhancements and changes in the latest editions are in direct response to your feedback, which I continue to welcome.

I write because I love to teach. While nothing compares to hands-on in-classroom instruction, turning those lessons into books that can be read far and wide has gifted me with expanding my teaching reach. It is thus a source of much gratification to see hundreds of colleges and universities use these SQL books as part of their IT and computer science curricula. Being included by professors and teachers in this way is both rewarding and humbling, and for that trust I am thankful.

And finally, thanks to the almost 1 million of you who bought the previous editions of these books (in over a dozen languages), making them not just my best-selling series but also the best-selling books on SQL. Your continued support is the highest compliment an author can ever be paid.

—Ben Forta

This page intentionally left blank

About the Author

Ben Forta is Adobe's Senior Director of Education Initiatives and has more than three decades of experience in the computer industry—in product development, support, training, and product marketing. He is the author of the best-selling *Sams Teach Yourself SQL in 10 Minutes* (as well as spinoff titles like this one and versions on SQL Server T-SQL, Oracle PL/SQL, and MariaDB), *Learning Regular Expressions*, and *Captain Code*, which teaches Python to younger coders (and those young at heart), Java, Windows, and more. He has extensive experience in database design and development, has implemented databases for several highly successful commercial software programs and websites, and is a frequent lecturer and columnist on application development and Internet technologies. Ben lives in Oak Park, Michigan, with his wife, Dr. Marcy Forta, and their children. He welcomes your email at ben@forta.com and invites you to visit his website at http://forta.com.

This page intentionally left blank

Introduction

MySQL is one of the most popular database management systems in the world. From small development projects to some of the best-known and most prestigious sites on the Web, MySQL has proven itself to be a solid, reliable, fast, and trusted solution for all sorts of data storage needs.

This book is based on my best-selling *Sams Teach Yourself SQL in 10 Minutes*. That book has become one of the most-used SQL tutorials in the world, with an emphasis on teaching what you really need to know—methodically, systematically, and simply. But as popular and as successful as that book is, it does have some limitations:

- In covering all of the major database management systems (DBMSs), coverage of DBMS-specific features and functionality had to be kept to a minimum.
- To simplify the SQL taught, the lowest common denominator had to be found—SQL statements that would (as much as possible) work with all major DBMSs. This requirement necessitated that better DBMS-specific solutions not be covered.
- Although basic SQL tends to be rather portable between DBMSs, more advanced SQL most definitely is not. As such, that book could not cover advanced topics, such as triggers, cursors, stored procedures, access control, and transactions, in any real detail.

And that is where this book comes in. *MySQL Crash Course* builds on the proven tutorials and structure of *Sams Teach Yourself SQL in 10 Minutes* without getting bogged down with anything except MySQL. Starting with simple data retrieval and working on to more complex topics, including the use of joins, subqueries, regular expression and full text-based searches, stored procedures, cursors, triggers, table constraints, and much more, you'll learn what you need to know methodically, systematically, and simply—in highly focused chapters designed to make you immediately and effortlessly productive.

When you turn to Chapter 1 and get to work, you'll be taking advantage of all MySQL has to offer in no time at all.

Who Is This Book For?

This book is for you if:

- You are new to SQL.
- You are just getting started with MySQL and want to hit the ground running.
- You want to quickly learn how to get the most out of MySQL.
- You want to learn how to use MySQL in your own application development.
- You want to be productive quickly and easily using MySQL without having to call someone for help.

Companion Website

This book has a companion website online at http://forta.com/books/9780138223021/. At this website, you'll find:

- The files used to create the example tables used throughout this book
- Answers to the questions in the "Challenges" section at the end of each chapter
- Online errata

Conventions Used in This Book

This book uses different typefaces to differentiate between code and regular English and also to help you identify important concepts.

Text that you type and text that should appear on your screen is presented in `monospace` type. `It looks like this to mimic the way text looks on your screen`.

Placeholders for variables and expressions appear in *`monospace italic`* font. You should replace a placeholder with the specific value it represents.

Note

A Note presents an interesting piece of information related to the surrounding discussion.

Tip

A Tip offers advice or teaches an easier way to do something.

Caution

A Caution advises you about potential problems and helps you steer clear of disaster.

New Term

A New Term box provides a clear definition of a new essential term.

Figure Credits

Figures 3.1-3.5: Oracle Corporation

► `Input`

The Input icon identifies code that you can type in yourself. It usually appears next to a listing.

► `Output`

The Output icon highlights the output produced by running MySQL code. It usually appears after input and next to output.

► `Analysis`

The Analysis icon alerts you to the line-by-line analysis of input or output.

This page intentionally left blank

1

Understanding SQL

In this chapter, you'll learn about databases and SQL, which are prerequisites to learning MySQL.

Database Basics

The fact that you are reading this book indicates that you, somehow, need to interact with databases, and MySQL specifically. And so, before diving into MySQL and its implementation of the SQL language, it is important that you understand some basic concepts about databases and database technologies.

Whether you are aware of it or not, you use databases all the time. Each time you select a name from your email address book, you are using a database. When you browse contacts on your phone, you are using a database. If you conduct a search on an Internet search site, you are using a database. When you log in to your network at work, you are validating your name and password against a database. Even when you use your ATM card at a cash machine, you are using databases for PIN verification and balance checking.

But even though we all use databases all the time, there remains much confusion over what exactly a database is. This is especially true because different people use the same database terms to mean different things. Therefore, a good place to start our study is with a list and explanation of the most important database terms.

Tip

Reviewing Basic Concepts What follows is a very brief overview of some basic database concepts. It is intended to either jolt your memory if you already have some database experience or to provide you with the absolute basics if you are new to databases. Understanding databases is an important part of mastering MySQL, and you might want to find a good book on database fundamentals to brush up on the subject, if needed.

What Is a Database?

The term *database* is used in many different ways, but for our purposes in this book, a database is a collection of data stored in some organized fashion. The simplest way to think of it is to imagine a database as a filing cabinet. The filing cabinet is simply a physical location to store data, regardless of what that data is or how it is organized.

New Term

Database A container (usually a file or set of files) for storing organized data.

Caution

Misuse Causes Confusion People often use the term *database* to refer to the database software they are running. This is incorrect, and it is a source of much confusion. Database software is actually called a *database management system* (or *DBMS*). A database is a container created and manipulated via a DBMS. A database might or might not be a file stored on a hard drive. And for the most part, this is not even significant as you never access a database directly anyway; you always use the DBMS, and it accesses the database for you.

Tables

When you store information in a filing cabinet, you don't just toss it in a drawer. Rather, you create files within the filing cabinet, and then you store related data in specific files.

In the database world, a file is called a *table*. A table is a structured file that can store data of a specific type. A table might contain a list of customers, a product catalog, or any other list of information.

New Term

Table A structured list of data of a specific type.

The key here is that the data stored in the table is one type of data or one list. You would never store a list of customers and a list of orders in the same database table. Doing so would make subsequent retrieval and access difficult. Rather, you'd create two tables, one for each list.

Every table in a database has a name that identifies it. That name is always unique—meaning no other table in that database can have the same name.

Note

Table Names What makes a table name unique is actually a combination of several things, including the database name and table name. While you cannot use the same table name twice in the same database, you definitely can reuse table names in different databases.

Tables have characteristics and properties that define how data is stored in them. These include information about what data may be stored, how it is broken up, how individual pieces of information are named, and much more. The set of information that describes a table is known as a *schema*, and a schema can be used to describe specific tables within a database, as well as an entire database (and the relationship between tables in a database, if any).

New Term

Schema Information about database and table layout and properties.

Note

Schema or Database? Occasionally the term *schema* is used as a synonym for *database* (and *schemata* as a synonym for *databases*). While unfortunate and frequently confusing, it is usually clear from the context which meaning of schema is intended. In this book, *schema* is used as defined here.

Columns and Datatypes

Tables are made up of columns. A column contains a particular piece of information within a table.

New Term

Column A single field in a table. Every table is made up of one or more columns.

The best way to understand this is to envision database tables as grids, somewhat like spreadsheets. Each column in the grid contains a particular piece of information. In a customer table, for example, the customer number is stored in one column, the customer name is stored in another, and the address, city, state, and zip code are all stored in their own columns.

Tip

Breaking Up Data It is extremely important to break data into multiple columns correctly. For example, city, state, and zip code should always be stored in separate columns. By breaking these out, it becomes possible to sort or filter data by specific columns (for example, to find all customers in a particular state or in a particular city). If city and state are combined into one column, it would be extremely difficult to sort or filter by state.

Each column in a database has an associated datatype. A datatype defines what type of data the column can contain. For example, if a column is to contain a number (perhaps the number of items in an order), it would be associated with the numeric datatype.

Columns that contain dates, text, notes, currency amounts, and so on would use the appropriate datatypes.

New Term

Datatype A type of allowed data. Every table column has an associated datatype that restricts (or allows) specific data in that column.

Datatypes restrict the type of data that can be stored in a column (for example, preventing the entry of alphabetical characters into a numeric field). Datatypes also help sort data correctly and play an important role in optimizing disk usage. As such, special attention must be given to picking the right datatype when tables are created.

Rows

Data in a table is stored in rows; each record saved is stored in its own row. Again, if you envision a table as a spreadsheet-style grid, the vertical columns in the grid are the table columns, and the horizontal rows are the table rows.

For example, a customers table might store one customer per row. The number of rows in the table is the number of records in the table.

New Term

Row A record in a table.

Note

Records or Rows? You might hear users refer to database *records* when referring to *rows*. For the most part, the two terms are used interchangeably, but *row* is technically the correct term.

Primary Keys

Every row in a table should have some column (or set of columns) that uniquely identifies it. A table containing customers might use a customer number column for this purpose, whereas a table containing orders might use the order ID. Similarly, an employee list table might use an employee ID column.

New Term

Primary Key A column (or set of columns) whose values uniquely identify every row in a table.

The column (or set of columns) that uniquely identifies each row in a table is called a *primary key*. The primary key is used to refer to a specific row. Without a primary key, updating or deleting specific rows in a table is extremely difficult because there is no guaranteed safe way to refer to just the rows that are affected.

Tip

Always Define Primary Keys Although primary keys are not actually required, most database designers ensure that every table they create has a primary key so that future data manipulation is possible and manageable.

Any column in a table can be established as the primary key, as long as it meets the following conditions:

- No two rows can have the same primary key value.
- Every row must have a primary key value. (Primary key columns may not allow NULL values.)

Note

Primary Key Rules The rules listed here are enforced by MySQL itself.

A primary key is usually defined on a single column within a table. But this is not required, and multiple columns may be used together as a primary key. When multiple columns are used, the rules previously listed must apply to all columns that make up the primary key, and the values of all columns together must be unique. (Individual columns need not have unique values.)

Tip

Primary Key Best Practices In addition to following the rules that MySQL enforces, you should adhere to several universally accepted best practices:

- Don't update values in primary key columns.
- Don't reuse values in primary key columns.
- Don't use values that might change in primary key columns. (For example, if you use a name as a primary key to identify a supplier and the supplier merges with another company and changes its name, you have to change the primary key.)

There is another very important type of key, called a *foreign key*, but I'll get to that in Chapter 15, "Joining Tables."

What Is SQL?

SQL (pronounced as the letters "S-Q-L" or as the word "sequel") is an abbreviation for Structured Query Language. SQL is a language designed specifically for communicating with databases.

Unlike other languages (spoken languages such as English or programming languages such as Python or Java), SQL is made up of very few words. This is deliberate. SQL is designed to do one thing and do it well: provide you with a simple and efficient way to read and write data from a database.

What are the advantages of SQL?

- SQL is not a proprietary language used by specific database vendors. Almost every major DBMS supports SQL, and learning this one language enables you to interact with just about every database you'll run into.
- SQL is easy to learn. The statements are all made up of descriptive English words, and there aren't that many of them.
- Despite its apparent simplicity, SQL is actually a very powerful language, and by cleverly using its language elements, you can perform very complex and sophisticated database operations.

Note

DBMS-Specific SQL Although SQL is not a proprietary language and there is a standards committee that tries to define SQL syntax that can be used by all DBMSs, the reality is that no two DBMSs implement SQL identically. The SQL taught in this book is specific to MySQL, and while much of the language taught will be usable with other DBMSs, you should not assume complete SQL syntax portability.

Try It Yourself

All of the chapters in this book use working examples, showing you the SQL syntax, what it does, and explaining why it does it. I strongly suggest that you try each and every example for yourself so that you learn MySQL firsthand.

In addition, starting in Chapter 4, "Retrieving Data," most chapters conclude with a "Challenges" section to help you review and gauge your MySQL proficiency. If you get stuck, you can go to the companion website to find the answers to the "Challenges" section questions.

Appendix B, "The Example Tables," describes the example tables used throughout this book and explains how to obtain and install them. If you have not done so yet, take a look at that appendix before proceeding.

Note

You Need MySQL Obviously, you'll need access to a copy of MySQL to follow along. Appendix A, "Getting Started with MySQL," explains where to get a copy of MySQL and provides some pointers for getting started. If you do not have access to a copy of MySQL, read that appendix before proceeding.

Summary

In this first chapter, you learned what SQL is and why it is useful. Because SQL is used to interact with databases, you also reviewed some basic database terminology.

This page intentionally left blank

2

Introducing MySQL

In this chapter, you'll learn what MySQL is, and you'll learn about the tools you can use when working with it.

What Is MySQL?

In the previous chapter, you learned about databases and SQL. As explained, it is the database software (the *DBMS* or *database management system*) that actually does all the work of storing, retrieving, managing, and manipulating data. MySQL is a DBMS; that is, it is database software.

MySQL has been around for a long time, and it is now installed and in use at millions of installations worldwide. Why do so many organizations and developers use MySQL? Here are some of the reasons:

- **Cost:** MySQL is open source, and it is usually free to use and even modify the software.
- **Performance:** MySQL is fast—make that very fast.
- **Trustworthiness:** MySQL is used by some of the most important and prestigious organizations and sites, all of which entrust it with their critical data.
- **Simplicity:** MySQL is easy to install and get up and running.

In fact, historically the only real criticism of MySQL has been that it has not always supported the functionality and features offered by other DBMSs. But as new features have been added to each new version, that has changed, and it continues to do so.

Client/Server Software

DBMSs fall into two categories: shared file–based and client/server DBMSs. Shared file–based DBMSs (which include products such as Microsoft Access and FileMaker) are designed for desktop use and are generally not intended for use on higher-end or more critical applications.

Databases such as MySQL, Oracle, and Microsoft SQL Server are client/server–based databases. Client/server applications are split into two distinct parts. The *server* portion is a piece of software that is responsible for all data access and manipulation. This software runs on a computer called the *database server*.

Only the server software interacts with the data files. All requests for data, data additions and deletions, and data updates are funneled through the server software. These requests or changes come from computers running client software. The *client* is the piece of software with which the user interacts. If you request an alphabetical list of products, for example, the client software submits that request over the network to the server software. The server software processes the request; filters, discards, and sorts data as necessary; and sends the results back to your client software.

Note

How Many Computers? The client and server software may be installed on two computers or on one computer. Either way, the client software communicates with the server software for all database interaction.

All this action occurs transparently to you, the user. The fact that data is stored elsewhere, or that a database server is performing all this processing for you, is hidden. You never need to access the data files directly. In fact, most networks are set up so that users have no access to the data or even the drives on which it is stored.

Why is this significant? Because to work with MySQL, you'll need access to both a computer running the MySQL server software and client software with which to issue commands to MySQL:

- The server software is the MySQL DBMS. You can be running a locally installed copy, or you can connect to a copy running on a remote server to which you have access.
- The client can be MySQL-provided tools (more on these below), scripting languages (such as Python and Ruby), web application development languages and platforms (such as ASP.NET, JavaScript, and Node.js), programming languages (such as C, C++, and Java), and more.

Note

Cloud-Based DBMSs You may have come across cloud-based DBMSs, which are hosted database services that are accessed from a web browser. These are technically client/server DBMSs; the client just happens to be code that runs in the browser.

MySQL Versions

We'll get back to client tools in a moment. First, a quick word about DBMS versions.

The current version of MySQL is version 8. Version 5 (all the way through 5.7) is still in use in many organizations. (Version 6 was never fully released, and there was no version 7.)

This book was written with MySQL version 8 in mind, but most of the chapters apply to version 5, too.

Note

Version Requirements Noted Any chapter that requires a specific version of MySQL clearly notes that at the start of the chapter.

MySQL Tools

As just explained, MySQL is a client/server DBMS, and so to use MySQL, you'll need a client—an application that you use to interact with MySQL (by giving it commands to be executed).

There are lots of client application options, but when learning MySQL (and, indeed, when writing and testing MySQL scripts), you are best off using a utility designed for just that purpose. And there are two tools in particular that warrant specific mention: the mysql command-line utility and the interactive MySQL Workbench.

`mysql` Command-Line Utility

Every MySQL installation comes with a simple command-line utility called `mysql`. This utility does not have any drop-down menus, fancy user interfaces, mouse support, or anything like that.

Typing `mysql` at your operating system command prompt brings up a simple prompt that looks like this:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.31 MySQL Community Server - GPL
Copyright (c) 2000, 2022, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

Note

MySQL Options and Parameters If you just type `mysql` by itself, you might receive an error message. If you do, it is likely because security credentials are needed or because MySQL is not running locally or on the default port. `mysql` accepts an array of command-line parameters you can (and might need to) use. For example, to specify the user login name `ben`, you use `mysql -u ben`. To specify a username, host name, and port and get a prompt for a password, you use `mysql -u ben -p -h myserver -P 9999`.

You can obtain a complete list of command-line options and parameters by using `mysql ---help`.

Of course, your version and connection information might differ, but you will be able to use this utility anyway. You'll note that:

- Commands are typed after the `mysql>` prompt.
- Commands end with ; or `\g`; in other words, just pressing Enter will not execute a command.
- Type `help` or `\h` to obtain help. You can also provide additional text to obtain help on specific commands (for example, `help select` to obtain help on using the `SELECT` statement).
- Type `quit` or `exit` to quit the command-line utility.

The `mysql` command-line utility is one of the most used utilities and is invaluable for quickly testing and executing scripts. In fact, all of the output examples used in this book were created with `mysql` command-line output.

Tip

Familiarize Yourself with the `mysql` Command-Line Utility Even if you opt to use the graphical tool described next, you should make sure to familiarize yourself with the `mysql` command-line utility, as this is the one client you can safely rely on to always be present (as it is part of the core MySQL installation).

MySQL Workbench

MySQL Workbench is a graphical interactive client used to write and execute MySQL commands.

MySQL Workbench consolidates and replaces several different interactive tools used in prior versions of MySQL. Since its release, this tool has quickly become a developer favorite.

Note

Obtaining MySQL Workbench Unlike the command-line utility `mysql`, MySQL Workbench is not always installed as part of the core MySQL DBMS installation. If it isn't, you can download it for free directly from https://dev.mysql.com/downloads/workbench/. (Versions are available for Linux, macOS, and Windows, and source code is downloadable, too.)

We'll start using MySQL Workbench in the next chapter as we explore connecting to and using the MySQL DBMS. For now, note the following:

- MySQL Workbench features a color-coded editor that can help you write SQL statements.
- You can test your SQL statements right inside MySQL Workbench, and results (if there are any) will be displayed in a grid right below your statements.

- MySQL Workbench also lists all available datasources (called schemas). You can expand any datasource to see its tables, and you can expand any table to see its columns.
- You can use MySQL Workbench to edit data, check the server status and settings, back up and restore data, and much more.

Tip

Execute Saved Scripts You can use MySQL Workbench to execute saved scripts. To do this, select File, Open SQL Script, select the script (which will be displayed in a new tab), and click the Execute button (the one with a yellow lightning bolt).

MySQL Workbench is an important tool, and I suggest that you use it for all chapters in this book. We'll look at it in detail in the next chapter.

Other Tools

You are not limited to using MySQL's provided clients. There are lots of third-party clients out there, too, and you can use any of them with MySQL. These are some popular ones:

- DBeaver
- HeidiSQL
- phpMyAdmin
- RazorSQL

To use any of these tools, you generally just need to provide server and login details.

Summary

In this chapter, you learned what exactly MySQL is. You were also introduced to two client utilities: an included command-line utility and an optional but highly recommended graphical utility.

This page intentionally left blank

3

Working with MySQL

In this chapter, you'll learn how to connect and log in to MySQL, how to issue MySQL statements, and how to obtain information about databases and tables.

Now that you have a MySQL DBMS and client software to use with it, it would be worthwhile to briefly discuss connecting to the database.

MySQL, like all other client/server DBMSs, requires that you log in to the DBMS before you can issue commands. Your login name might not be the same as your network login name (assuming that you are using a network); MySQL maintains its own list of users internally and associates rights with each user.

When you first installed MySQL, you were probably prompted for an administrative login (often `root`) and a password. If you are using your own local server and are simply experimenting with MySQL, using that login is fine. In the real world, however, the administrative login is closely protected because access to it grants full rights to create tables, drop entire databases, change logins and passwords, and more.

To connect to MySQL, you need the following pieces of information:

- The hostname (the name of the computer), which is `localhost` if you're connecting to a local MySQL server
- The port (if a port other than the default `3306` is used)
- A valid username
- The user password (if required)

As explained in Chapter 2, "Introducing MySQL," all of this information can be passed to the `mysql` command-line utility or entered into the server connection screen in MySQL Workbench.

Note

Using Other Clients If you are using a client other than the ones mentioned here, you still need to provide this information in order to connect to MySQL.

Using the Command-Line Tool

The `mysql` command-line utility is the one client tool that will always be available to you. Even though you are most likely going to use MySQL Workbench as you

learn MySQL with this book, it's worth learning how to use the command-line utility, too.

To start using the command-line tool with a locally installed MySQL DBMS, you can use this command:

```
mysql --user=root –-password
```

If you are not using the `root` user, specify your username instead. You can specify the password on the command line, or if you just specify `--password`, you'll be prompted to type it after you press Enter.

You then see text that looks something like this:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 31
Server version: 8.0.31 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

You enter commands at the `mysql>` prompt.

Selecting a Database

When you first connect to MySQL, you do not have any databases open for use. Before you can perform any database operations, you need to select a database. To do this, you use the `USE` keyword.

New Term

Keyword A reserved word that is part of the MySQL language. Never name a table or column using a keyword. Appendix E, "MySQL Reserved Words," lists the MySQL keywords.

For example, to use the `mysql` database, you would enter the following:

► `Input`

```
USE mysql;
```

► `Output`

```
Database changed
```

► `Analysis`

The `USE` statement does not return any results. Depending on the client used, some form of notification might be displayed. For example, the `mysql` command-line utility displays the `Database changed` message shown here upon successful database selection.

Remember that you must always "`USE` a database" before you can access any data in it.

Learning About Databases and Tables

What if you don't know the names of the available databases? And, for that matter, how is MySQL Workbench able to display a list of available databases?

Information about databases, tables, columns, users, privileges, and more is stored in databases and tables themselves. (Yes, MySQL uses MySQL to store this information.) But these internal tables are generally not accessed directly. Instead, you use the MySQL `SHOW` command to display this information (which MySQL extracts from those internal tables). Look at the following example:

► Input

```
SHOW DATABASES;
```

► Output

```
+--------------------+
| Database           |
+--------------------+
| crashcourse        |
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
7 rows in set (0.00 sec)
```

► Analysis

`SHOW DATABASES;` returns a list of available databases. This list might include databases used by MySQL internally (such as `mysql` and `information_schema` in this example). Of course, your own list of databases might not look like the one shown here.

To obtain a list of tables in a database, use `SHOW TABLES;`, as in this example:

► Input

```
SHOW TABLES;
```

► Output

```
+----------------------------------------------------+
| Tables_in_mysql                                    |
+----------------------------------------------------+
| columns_priv                                       |
| component                                          |
| db                                                 |
```

```
| default_roles                                          |
| engine_cost                                            |
| func                                                   |
| general_log                                            |
| global_grants                                          |
| gtid_executed                                          |
| help_category                                          |
| help_keyword                                           |
| help_relation                                          |
| help_topic                                             |
| innodb_index_stats                                     |
| innodb_table_stats                                     |
| ndb_binlog_index                                       |
| password_history                                       |
| plugin                                                 |
| procs_priv                                             |
| proxies_priv                                           |
| replication_asynchronous_connection_failover           |
| replication_asynchronous_connection_failover_managed   |
| replication_group_configuration_version                |
| replication_group_member_actions                       |
| role_edges                                             |
| server_cost                                            |
| servers                                                |
| slave_master_info                                      |
| slave_relay_log_info                                   |
| slave_worker_info                                      |
| slow_log                                               |
| tables_priv                                            |
| time_zone                                              |
| time_zone_leap_second                                  |
| time_zone_name                                         |
| time_zone_transition                                   |
| time_zone_transition_type                              |
| user                                                   |
+--------------------------------------------------------+
38 rows in set (0.00 sec)
+----------------------+
```

Note

Use of the terms master/slave is ONLY in association with the official terminology used in industry specifications and standards, and in no way diminishes Pearson's commitment to promoting diversity, equity, and inclusion, and challenging, countering and/or combating bias and stereotyping in the global population of the learners we serve.

► Analysis

`SHOW TABLES;` returns a list of available tables in the currently selected database. You can also use `SHOW` to display a table's columns:

Input

```
SHOW COLUMNS FROM servers;
```

Output

```
+-------------+-----------+------+-----+---------+-------+
| Field       | Type      | Null | Key | Default | Extra |
+-------------+-----------+------+-----+---------+-------+
| Server_name | char(64)  | NO   | PRI |         |       |
| Host        | char(255) | NO   |     |         |       |
| Db          | char(64)  | NO   |     |         |       |
| Username    | char(64)  | NO   |     |         |       |
| Password    | char(64)  | NO   |     |         |       |
| Port        | int       | NO   |     | 0       |       |
| Socket      | char(64)  | NO   |     |         |       |
| Wrapper     | char(64)  | NO   |     |         |       |
| Owner       | char(64)  | NO   |     |         |       |
+-------------+-----------+------+-----+---------+-------+
9 rows in set (0.00 sec)
```

Tip

The `DESCRIBE` Statement MySQL supports the use of `DESCRIBE` as a shortcut for `SHOW COLUMNS FROM`. In other words, `DESCRIBE customers;` is a shortcut for `SHOW COLUMNS FROM customers;`.

Other `SHOW` statements are supported, too, including these:

- `SHOW STATUS` is used to display extensive server status information.
- `SHOW CREATE DATABASE` and `SHOW CREATE TABLE` are used to display the MySQL statements used to create specified databases or tables, respectively.
- `SHOW GRANTS` is used to display security rights granted to users (all users or a specific user).
- `SHOW ERRORS` and `SHOW WARNINGS` are used to display server error or warning messages.

Note that client applications use these same MySQL commands. Applications that display interactive lists of databases and tables, that allow for the interactive creation and editing of tables, that facilitate data entry and editing, and that allow for user account and rights management all accomplish what they do by using the same MySQL commands that you can execute directly yourself.

Tip

Learning More About `SHOW` With the `mysql` command-line utility, you can execute the command `HELP SHOW;` to display a list of allowed `SHOW` statements.

Using MySQL Workbench

The `mysql` command-line utility is an invaluable—and readily available—tool. But it's not exactly intuitive or friendly to use. Fortunately, there is a wonderful interactive tool called MySQL Workbench that we'll be using throughout this book.

Tip

Obtaining MySQL Workbench MySQL Workbench is supported on Windows, Mac, and Linux. Refer to Appendix A, "Getting Started with MySQL," for help obtaining MySQL Workbench.

Getting Started

MySQL Workbench needs to know what DBMS you will be using. When MySQL Workbench first runs, it looks for a local DBMS, and if it finds one, it automatically adds the DBMS to the MySQL Connections list, as shown in Figure 3.1.

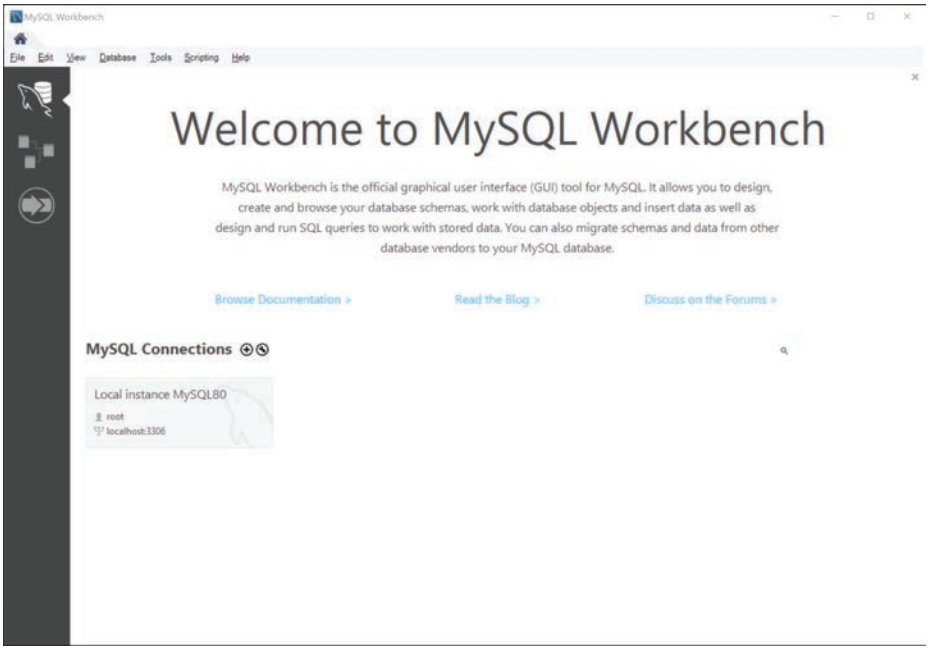

Figure 3.1 DBMS listed under MySQL Connections.

Double-click on the connection, and you are prompted for the password, as shown in Figure 3.2.

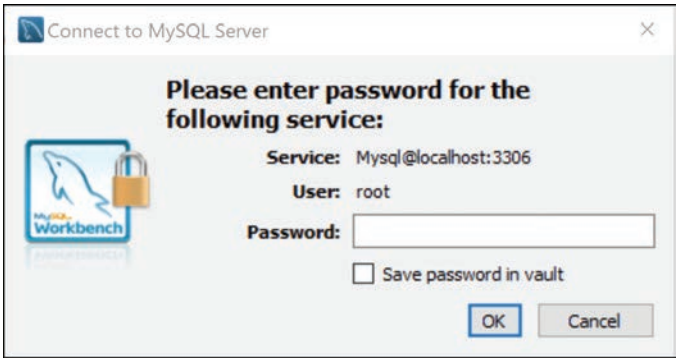

Figure 3.2 The password prompt.

If the login information you enter is correct, you're then connected to the MySQL server.

Using MySQL Workbench

Let's take a few minutes to become familiar with the MySQL Workbench user interface, shown in Figure 3.3.

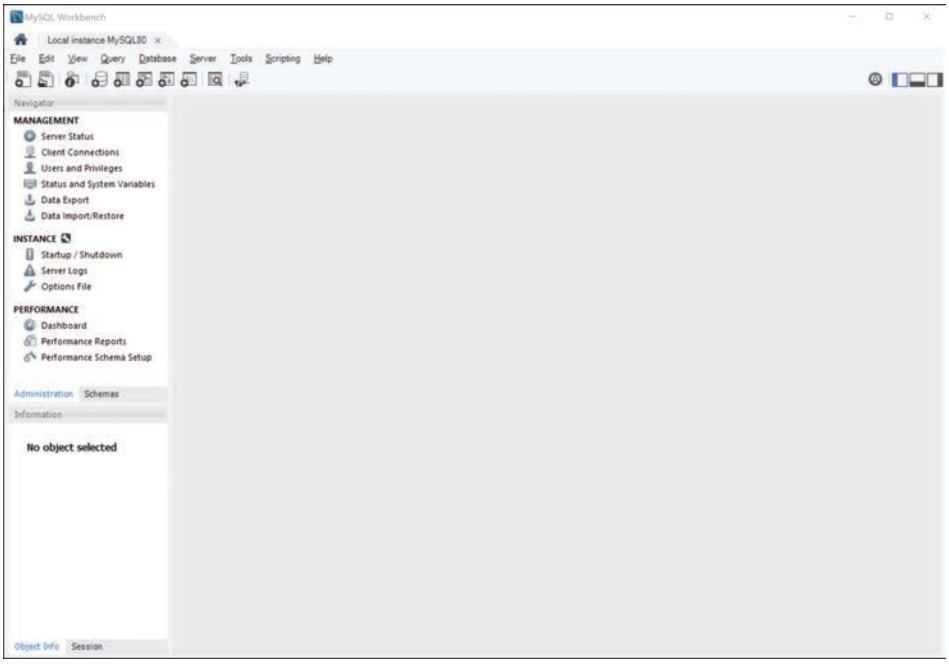

Figure 3.3 The MySQL Workbench user interface.

The upper left is the Navigator. It provides options to manage the server, including displaying server status (much the same as the information displayed previously using the command-line tool).

If you click on the Schemas tab in the Navigator, you can see all of your databases.

Selecting a Database

To select a database in MySQL Workbench, simple double-click it in the Schemas tab. The database is expanded, and its name becomes bold to indicate that it's ready for use.

Note

It's Using `USE` Earlier you learned about the `USE` statement, which opens a database. When you double-click on a database in MySQL Workbench, the tool is using a `USE` statement for you automatically. In fact, every option in MySQL Workbench uses SQL statements for you.

Learning About Databases and Tables

To see details about databases and tables, simply click on them. Details are displayed in the Information panel beneath the Navigation panel, as shown in Figure 3.4.

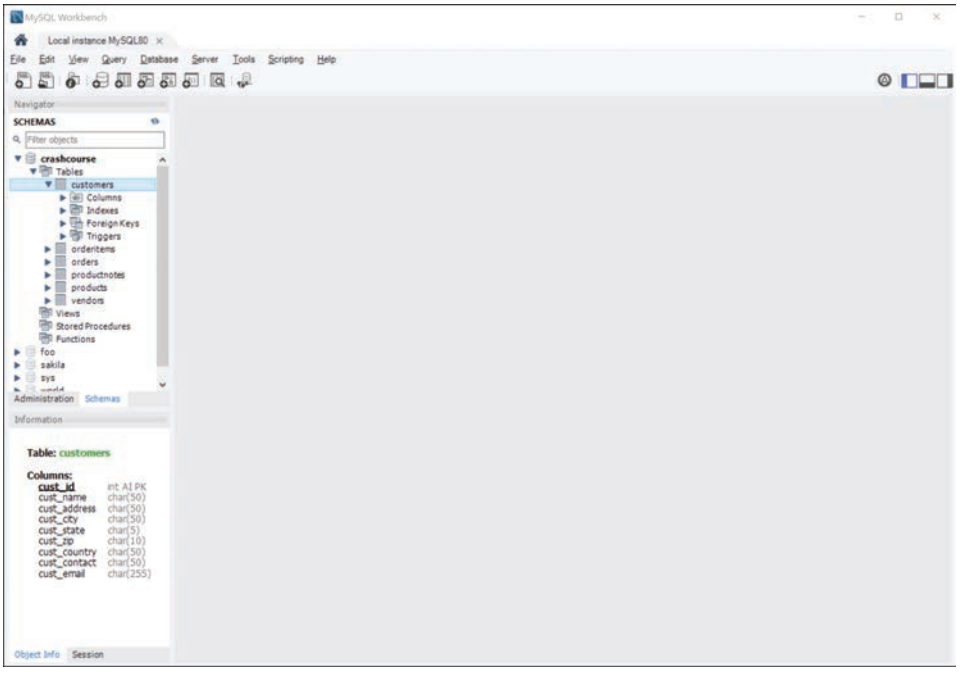

Figure 3.4 The Information panel below the Navigation panel.

Executing SQL Statements

Throughout this book, you'll be writing and testing SQL statements, and it's important to know how to do that with MySQL Workbench.

With a database open (any database will do), click the leftmost button on the toolbar; this is the New Query button, and it will open an editor window where you can type your SQL commands.

Figure 3.5 shows the database `world` being used, with the following SQL statement entered:

```
SELECT * FROM country;
```

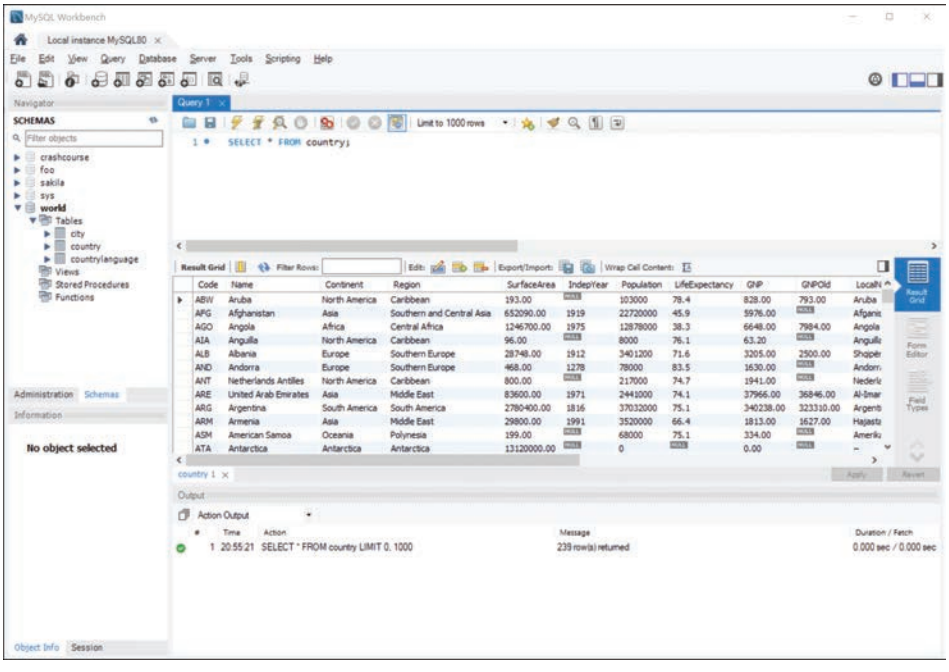

Figure 3.5 Query results are displayed in a grid below the SQL statement.

To execute (or run) the SQL statement, click the yellow lightning bolt button above the query window. The SQL statement is then executed, and results (if there are any) are displayed in a grid below.

Now you're ready to install the example database and tables and proceed with learning MySQL.

Next Steps

Now that you know how to connect and log in to your MySQL DBMS, you are ready to create the example database and tables used throughout this book.

Refer to Appendix B, "The Example Tables," for a detailed description of the tables as well as step-by-step instructions for installing them.

Summary

In this chapter, you learned how to connect and log in to MySQL and how to execute SQL commands by using both the command-line utility and MySQL Workbench. Armed with this knowledge, you can now dig into the all-important `SELECT` statement.

4

Retrieving Data

In this chapter, you'll learn how to use the `SELECT` statement to retrieve one or more columns of data from a table.

The `SELECT` Statement

As explained in Chapter 1, "Understanding SQL," SQL statements are made up of plain English terms. These terms are called *keywords*, and every SQL statement is made up of one or more keywords. The SQL statement you'll probably use most frequently is the `SELECT` statement. Its purpose is to retrieve information from one or more tables.

To use `SELECT` to retrieve table data, you must, at a minimum, specify two pieces of information: what you want to select and from where you want to select it.

Tip

Use MySQL Workbench to Follow Along As noted previously, you are strongly encouraged to try each and every example as you work through this book. Actually, you should also experiment and tweak the SQL yourself. The more you do so, the more comfortable you'll become with MySQL syntax. Use MySQL Workbench to do this, as explained in Chapter 3, "Working with MySQL."

Retrieving Individual Columns

We'll start with a simple `SQL SELECT` statement, as follows:

► `Input`

```
SELECT prod_name
FROM products;
```

► `Analysis`

This statement uses the `SELECT` statement to retrieve a single column called `prod_name` from the `products` table. The desired column name is specified right after the `SELECT`

keyword, and the `FROM` keyword specifies the name of the table from which to retrieve the data. This statement provides the following output:

Output

```
+------------------ +
| prod_name         |
+------------------ +
|.5 ton anvil       |
|1 ton anvil        |
|2 ton anvil        |
|Oil can            |
|Fuses              |
|Sling              |
|TNT (1 stick)      |
|TNT (5 sticks)     |
|Bird seed          |
|Carrots            |
|Safe               |
|Detonator          |
|JetPack 1000       |
|JetPack 2000       |
+------------------ +
```

Note

Unsorted Data If you tried this query yourself (you did, right?), you might have discovered that the data was displayed in a different order than shown here. If this is the case, don't worry; it is working exactly as it is supposed to. If query results are not explicitly sorted (we'll get to that in the next chapter), data will be returned in no order of any significance. It might be the order in which the data was added to the table, but it might not. As long as your query returned the same number of rows as shown here, then it is working.

A simple `SELECT` statement like the one just shown returns all the rows in a table. Data is not filtered (to retrieve a subset of the results), nor is it sorted. We'll discuss these topics in the next few chapters.

Note

Terminating Statements Multiple SQL statements must be separated by semicolons (; characters). MySQL (like most other DBMSs) does not require that a semicolon be used after a single statement. You can always add a semicolon if you wish. It'll do no harm, even if it isn't needed.

If you are using the `mysql` command-line client, the semicolon is always needed (as explained in Chapter 2, "Introducing MySQL").

Note

SQL Statements and Case It is important to note that SQL statements are not case-sensitive, so `SELECT` is the same as `select` or `Select`. Many SQL developers find that using uppercase for all SQL keywords and lowercase for column and table names makes code easier to read and debug. (That's the formatting style used in this book.)

However, be aware that while the SQL language is not case-sensitive, identifiers (the names of databases, tables, and columns) might be.

As a best practice, pick a case convention and use it consistently.

Tip

Use of White Space All extra white space within a SQL statement is ignored when the statement is processed. SQL statements can be specified on one long line or broken up over many lines. The following statements are thus functionally identical:

```
SELECT prod_name FROM products;

SELECT
prod_name
FROM
products;

        SELECT
           prod_name
              FROM
                 products;
```

Most SQL developers find that breaking up statements over multiple lines makes the statements easier to read and debug.

White space (spaces, tabs, line breaks, etc.) is ignored when SQL statements are processed. So how does the DBMS know when your statement is finished? That's what the ; at the end does.

Retrieving Multiple Columns

To retrieve multiple columns from a table, you use the same `SELECT` statement. The only difference is that you must specify multiple column names after the `SELECT` keyword, and you must separate the columns with commas.

Tip

Take Care with Commas When selecting multiple columns, be sure to include a comma after each column name except the last one. Including a comma after the last name will generate an error.

The following `SELECT` statement retrieves three columns from the `products` table:

► Input

```
SELECT prod_id, prod_name, prod_price
FROM products;
```

► Analysis

Just as in the previous example, this statement uses the `SELECT` statement to retrieve data from the `products` table. In this example, three column names are specified, separated by commas. The output from this statement is as follows:

► Output

```
+---------+---------------+------------+
| prod_id | prod_name     | prod_price |
+---------+---------------+------------+
| ANV01   | .5 ton anvil  | 5.99       |
| ANV02   | 1 ton anvil   | 9.99       |
| ANV03   | 2 ton anvil   | 14.99      |
| OL1     | Oil can       | 8.99       |
| FU1     | Fuses         | 3.42       |
| SLING   | Sling         | 4.49       |
| TNT1    | TNT (1 stick) | 2.50       |
| TNT2    | TNT (5 sticks)| 10.00      |
| FB      | Bird seed     | 10.00      |
| FC      | Carrots       | 2.50       |
| SAFE    | Safe          | 50.00      |
| DTNTR   | Detonator     | 13.00      |
| JP1000  | JetPack 1000  | 35.00      |
| JP2000  | JetPack 2000  | 55.00      |
+---------+---------------+------------+
```

Note

Presentation of Data SQL statements typically return raw, unformatted data. Data formatting is a presentation issue, not a retrieval issue. Therefore, presentation (for example, alignment and displaying the price values as currency amounts with the currency symbol and commas) is typically specified in the application that displays the data. Actual raw retrieved data (without application-provided formatting) is rarely displayed as is.

Retrieving All Columns

In addition to being able to specify one or more columns, `SELECT` statements can also request all columns without having to list them individually. This is done using the asterisk (*) wildcard character in lieu of actual column names, as follows:

► Input

```
SELECT *
FROM products;
```

► Analysis

When a wildcard (*) is specified, all the columns in the table are returned. Columns are returned in the order in which they appear in the table definition. However, changes to table schemas (such as adding and removing columns) may cause ordering changes.

Caution

Using Wildcards As a rule, you are better off not using the * wildcard unless you really do need every column in the table. Even though using wildcards might save you the time and effort needed to list the desired columns explicitly, retrieving unnecessary columns usually slows down the performance of your retrieval and your application.

Tip

Retrieving Unknown Columns There is one big advantage to using wildcards. As you do not explicitly specify column names (because the asterisk retrieves every column), it is possible to retrieve columns whose names are unknown.

Retrieving Distinct Rows

As you have seen, `SELECT` returns all matched rows. But what if you did not want every occurrence of every value? For example, suppose you wanted the vendor ID of every vendor with products in your `products` table and used the following:

► Input

```
SELECT vend_id
FROM products;
```

► Output

```
+---------+
| vend_id |
+---------+
|    1001 |
|    1001 |
|    1001 |
```

```
|    1002 |
|    1002 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1005 |
|    1005 |
+---------+
```

► Analysis

This `SELECT` statement returns 14 rows because there are 14 products listed in the `products` table. But those 14 products are actually sold by just 4 vendors. So how could you retrieve a list of distinct vendor values?

The solution is to use the `DISTINCT` keyword which, as its name implies, instructs MySQL to return only distinct values:

► Input

```
SELECT DISTINCT vend_id
FROM products;
```

► Analysis

`SELECT DISTINCT vend_id` tells MySQL to return only distinct (unique) `vend_id` rows, and so only 4 rows are returned, as shown in the following output. When you use the `DISTINCT` keyword, you must place it directly in front of the column names.

► Output

```
+--------- +
| vend_id  |
+--------- +
|     1001 |
|     1002 |
|     1003 |
|     1005 |
+--------- +
```

Caution

Can't Be Partially `DISTINCT` The `DISTINCT` keyword applies to all columns, not just the one it precedes. If you were to specify `SELECT DISTINCT vend_id, prod_price`, all rows would be retrieved unless *both* of the specified columns were distinct.

Limiting Results

The SELECT statement returns all matched rows—and possibly every row—in the specified table. To return just the first row or rows, use the LIMIT clause. Here is an example:

► Input

```
SELECT prod_name
FROM products
LIMIT 5;
```

► Analysis

This example uses the SELECT statement to retrieve a single column. LIMIT 5 instructs MySQL to return no more than 5 rows. The output from this statement is as follows:

► Output

```
+---------------- +
| prod_name       |
+---------------- +
| .5 ton anvil    |
| 1 ton anvil     |
| 2 ton anvil     |
| Oil can         |
| Fuses           |
+---------------- +
```

To get the next 5 rows, specify both where to start and the number of rows to retrieve, like this:

► Input

```
SELECT prod_name
FROM products
LIMIT 5,5;
```

► Analysis

LIMIT 5,5 instructs MySQL to return 5 rows, starting from row 5. The first number is where to start, and the second is the number of rows to retrieve. The output from this statement is as follows:

► Output

```
+---------------- +
| prod_name       |
+---------------- +
| Sling           |
| TNT (1 stick)   |
| TNT (5 sticks)  |
| Bird seed       |
| Carrots         |
+---------------- +
```

So, LIMIT with one value specified always starts from the first row, and the specified number is the number of rows to return. LIMIT with two values specified can start from wherever that first value tells it to.

Caution

Row 0 The first row retrieved is row 0, not row 1. Therefore, `LIMIT 1,1` will retrieve the second row, not the first one.

Note

When There Aren't Enough Rows The number of rows to retrieve specified in LIMIT is the *maximum* number to retrieve. If there aren't enough rows (for example, if you specify `LIMIT 10,5`, but there are only 13 rows), MySQL returns as many as it can.

Tip

Using `LIMIT OFFSET` Does `LIMIT 3,4` mean 3 rows starting from row 4 or 4 rows starting from row 3? As you just learned, it means 4 rows starting from row 3, but it is a bit ambiguous.

For this reason, MySQL supports an alternative syntax for LIMIT. `LIMIT 4 OFFSET 3` means to get 4 rows starting from row 3, just like `LIMIT 3,4`. `LIMIT OFFSET` tends not to be used much, and the reason I am mentioning it here is so that you'll know what it is if you see it in the wild.

Using Fully Qualified Table Names

The SQL examples used thus far have referred to columns by using just the column names. It is also possible to refer to columns by using fully qualified names (that is, using both the table and column names). Look at this example:

Input

```
SELECT products.prod_name
FROM products;
```

Analysis

This SQL statement is functionally identical to the very first one used in this chapter, but here a fully qualified column name is specified. It explicitly states that the `prod_name` column it's referring to is in the `products` table.

Table names, too, may be fully qualified, as shown here:

Input

```
SELECT products.prod_name
FROM crashcourse.products;
```

Analysis

Once again, this statement is functionally identical to the one just used (assuming, of course, that the `products` table is indeed in the `crashcourse` database).

There are situations when fully qualified names are required, as you will see in later chapters. For now, it is worth noting this syntax so you'll know what it is if you run across it.

Using Comments

As you have seen, MySQL statements are instructions that are processed by the MySQL DBMS. But what if you want to include text that you do not want to be processed and executed? Why would you ever want to do that? Here are a few reasons:

- The SQL statements we've been using here are all very short and very simple. But, as your SQL statements grow in length and complexity, you'll want to include descriptive comments (for your own future reference or for whoever must work on the project next). These comments need to be embedded in the SQL scripts, but they are obviously not intended for actual DBMS processing.
- The same is true for introductory text at the top of a SQL file, which you'll often want to contain a description and notes and perhaps even programmer contact information.
- Another important use for comments is to temporarily stop SQL code from being executed. If you are working with a long SQL statement and want to test just part of it, you can *comment out* some of the code so that the DBMS sees it as comments and ignores it.

MySQL supports several forms of comment syntax. We'll start with inline comments:

Input

```
SELECT prod_name    -- this is a comment
FROM Products;
```

Analysis

Comments may be embedded inline using `--` (two hyphens). Any text on the same line that is after the `--` is considered comment text, making this a good option for describing columns in a `CREATE TABLE` statement, for example.

Here is another form of inline comment:

Input

```
# This is a comment
SELECT prod_name
FROM Products;
```

Analysis

`#` can be used anywhere in your SQL. Anything that follows this character is comment text, so `#` at the start of a line makes the entire line a comment.

You can also create multiline comments and comments that stop and start anywhere within the script:

Input

```
/* SELECT prod_name, vend_id
FROM Products; */
SELECT prod_name
FROM Products;
```

Analysis

`/*` starts a comment, and `*/` ends it. Anything between `/*` and `*/` is comment text. This type of comment is often used to *comment out* code, as shown in this example. Here, two `SELECT` statements are defined, but the first one won't execute because it has been commented out.

Summary

In this chapter, you learned how to use the SQL `SELECT` statement to retrieve a single table column, multiple table columns, and all the columns in a table. You also learned how (and why) to comment your SQL code. Next, you'll learn how to sort the retrieved data.

Challenges

1. Write a SQL statement to retrieve every customer ID (`cust_id`) from the `customers` table.
2. The `OrderItems` table contains every item ordered (some of which were ordered multiple times). Write a SQL statement to retrieve a list of the products (`prod_id`) ordered (not every order, just a unique list of products). Here's a hint: You should end up with nine unique rows displayed.
3. Write a SQL statement that retrieves all columns from the `customers` table and an alternate `SELECT` statement that retrieves just the customer IDs. Use comments to comment out one `SELECT` statement so you can run the other one. (And, of course, test both statements.)

5

Sorting Retrieved Data

In this chapter, you will learn how to use the `SELECT` statement's `ORDER BY` clause to sort retrieved data as needed.

Sorting Data

As you learned in the previous chapter, the following SQL statement returns a single column from a database table. But look at the output. The data appears to be displayed in no particular order.

► Input

```
SELECT prod_name
FROM products;
```

► Output

```
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Oil can        |
| Fuses          |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
| Bird seed      |
| Carrots        |
| Safe           |
| Detonator      |
| JetPack 1000   |
| JetPack 2000   |
+----------------+
```

Actually, the retrieved data is not displayed in a merely random order. Unsorted data is typically displayed in the order in which it appears in the underlying tables. This could

be the order in which the data was added to the tables initially. However, if data was subsequently updated or deleted, the order is affected by how MySQL reuses reclaimed storage space. The end result is that you cannot (and should not) rely on the sort order if you do not explicitly control it. Relational database design theory states that the sequence of retrieved data cannot be assumed to have significance if ordering was not explicitly specified.

New Term

Clauses These are parts of SQL statements are made up of clauses, some required and some optional. A clause usually consists of a keyword and supplied data. An example of this is the `SELECT` statement's `FROM` clause, which you saw in the previous chapter.

To explicitly sort data retrieved using a `SELECT` statement, you use the `ORDER BY` clause. `ORDER BY` takes the name of one or more columns by which to sort the output. Look at the following example:

Input

```
SELECT prod_name
FROM products
ORDER BY prod_name;
```

Analysis

This statement is identical to the earlier statement, except that it also specifies an `ORDER BY` clause instructing MySQL to sort the data alphabetically based on the `prod_name` column. The results are as follows:

Output

```
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Bird seed      |
| Carrots        |
| Detonator      |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
```

Tip

Sorting by Nonselected Columns More often than not, the columns used in an `ORDER BY` clause are ones that were selected for display. However, this is actually not required, and it is perfectly legal to sort data based on a column that is not retrieved.

Sorting by Multiple Columns

It is often necessary to sort data by more than one column. For example, if you are displaying an employee list, you might want to display it sorted by last name and first name (that is, first sort by last name and then, within each last name, sort by first name). This is useful, for example, if there are multiple employees with the same last name.

To sort by multiple columns, simply specify the column names separated by commas, just as you do when you are selecting multiple columns.

The following code retrieves three columns and sorts the results by two of them—first by price and then by name:

Input

```
SELECT prod_id, prod_price, prod_name
FROM products
ORDER BY prod_price, prod_name;
```

Output

```
+---------+------------+----------------+
| prod_id | prod_price | prod_name      |
+---------+------------+----------------+
| FC      |       2.50 | Carrots        |
| TNT1    |       2.50 | TNT (1 stick)  |
| FU1     |       3.42 | Fuses          |
| SLING   |       4.49 | Sling          |
| ANV01   |       5.99 | .5 ton anvil   |
| OL1     |       8.99 | Oil can        |
| ANV02   |       9.99 | 1 ton anvil    |
| FB      |      10.00 | Bird seed      |
| TNT2    |      10.00 | TNT (5 sticks) |
| DTNTR   |      13.00 | Detonator      |
| ANV03   |      14.99 | 2 ton anvil    |
| JP1000  |      35.00 | JetPack 1000   |
| SAFE    |      50.00 | Safe           |
| JP2000  |      55.00 | JetPack 2000   |
+---------+------------+----------------+
```

It is important to understand that when you are sorting by multiple columns, the sort sequence is exactly as specified. In other words, using the output in this example, the products are sorted based on the `prod_name` column only when multiple rows have the same `prod_price` value. If all the values in the `prod_price` column are unique, no data is sorted by `prod_name`.

Sorting by Column Position

In addition to being able to specify sort order by using column names, you can also use `ORDER BY` to order based on relative column position. The best way to understand this is to look at an example:

► Input

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY 2, 3;
```

► Output

```
prod_id    prod_price    prod_name
-------    ----------    --------------------
BNBG02     3.4900        Bird bean bag toy
BNBG01     3.4900        Fish bean bag toy
BNBG03     3.4900        Rabbit bean bag toy
RGAN01     4.9900        Raggedy Ann
BR01       5.9900        8 inch teddy bear
BR02       8.9900        12 inch teddy bear
RYL01      9.4900        King doll
RYL02      9.4900        Queen doll
BR03       11.9900       18 inch teddy bear
```

► Analysis

As you can see, the output here is identical to that of the query above. The difference here is in the `ORDER BY` clause. Instead of specifying column names, you specify the relative positions of selected columns in the `SELECT` list. `ORDER BY 2` means sort by the second column in the `SELECT` list, the `prod_price` column. `ORDER BY 2, 3` means sort by `prod_price` and then by `prod_name`.

The primary advantage of this technique is that it doesn't require you to retype the column names. But there are some downsides, too. First, not explicitly listing column names increases the likelihood of mistakenly specifying the wrong column. Second, it is easy to mistakenly reorder data when making changes to the `SELECT` list (such as when you forget to make the corresponding changes to the `ORDER BY` clause). And finally, obviously you cannot use this technique when sorting by columns that are not in the `SELECT` list.

Tip

Sorting by Nonselected Columns You cannot use `ORDER BY` to order based on relative column position when sorting by columns that do not appear in the `SELECT` list. However, you can mix and match column names and relative column positions in a single statement, if needed.

Specifying Sort Direction

Data sorting is not limited to ascending sort orders (that is, from A to Z). Although this is the default sort order, you can also use the `ORDER BY` clause to sort in descending order (that is, from Z to A). To sort in descending order, you need to specify the `DESC` keyword.

The following example sorts products by price, in descending order (that is, most expensive first):

Input

```
SELECT prod_id, prod_price, prod_name
FROM products
ORDER BY prod_price DESC;
```

Output

```
+---------+------------+----------------+
| prod_id | prod_price | prod_name      |
+---------+------------+----------------+
| JP2000  |      55.00 | JetPack 2000   |
| SAFE    |      50.00 | Safe           |
| JP1000  |      35.00 | JetPack 1000   |
| ANV03   |      14.99 | 2 ton anvil    |
| DTNTR   |      13.00 | Detonator      |
| TNT2    |      10.00 | TNT (5 sticks) |
| FB      |      10.00 | Bird seed      |
| ANV02   |       9.99 | 1 ton anvil    |
| OL1     |       8.99 | Oil can        |
| ANV01   |       5.99 | .5 ton anvil   |
| SLING   |       4.49 | Sling          |
| FU1     |       3.42 | Fuses          |
| FC      |       2.50 | Carrots        |
| TNT1    |       2.50 | TNT (1 stick)  |
+---------+------------+----------------+
```

But what if you were to sort by multiple columns? The following example sorts the products in descending order (that is, most expensive first) and based on product name:

Input

```
SELECT prod_id, prod_price, prod_name
FROM products
ORDER BY prod_price DESC, prod_name;
```

Output

```
+---------+------------+----------------+
| prod_id | prod_price | prod_name      |
+---------+------------+----------------+
| JP2000  |      55.00 | JetPack 2000   |
| SAFE    |      50.00 | Safe           |
| JP1000  |      35.00 | JetPack 1000   |
```

```
| ANV03   |      14.99 | 2 ton anvil    |
| DTNTR   |      13.00 | Detonator      |
| FB      |      10.00 | Bird seed      |
| TNT2    |      10.00 | TNT (5 sticks) |
| ANV02   |       9.99 | 1 ton anvil    |
| OL1     |       8.99 | Oil can        |
| ANV01   |       5.99 | .5 ton anvil   |
| SLING   |       4.49 | Sling          |
| FU1     |       3.42 | Fuses          |
| FC      |       2.50 | Carrots        |
| TNT1    |       2.50 | TNT (1 stick)  |
+---------+------------+----------------+
```

► Analysis

The `DESC` keyword only applies to the column name that directly precedes it. In this example, `DESC` is specified for the `prod_price` column but not for the `prod_name` column. Therefore, the `prod_price` column is sorted in descending order, but the `prod_name` column (within each price) is still sorted in standard ascending order.

Tip

Sorting Descending on Multiple Columns If you want to sort descending on multiple columns, be sure each column has its own `DESC` keyword.

The opposite of `DESC` is `ASC` (for *ascending*), which you can specify to sort in ascending order. In practice, however, `ASC` is not usually used because ascending order is the default sequence (and is assumed if neither `ASC` nor `DESC` is specified).

Tip

Case-Sensitivity and Sort Orders When you are sorting textual data, is `A` the same as `a`? And does `a` come before `B` or after `Z`? These are not theoretical questions, and the answers depend on how the database is set up.

In *dictionary* sort order, `A` is treated the same as `a`, and that is the default behavior in MySQL (and, indeed, in most other DBMSs as well). However, administrators can change this behavior, if needed. (If your database contains lots of foreign language characters, for example, this might become necessary.)

The key here is that if you need an alternate sort order, you cannot accomplish it with a simple `ORDER BY` clause. You must contact your database administrator.

By using a combination of `ORDER BY` and `LIMIT`, it is possible to find the highest or lowest value in a column. The following example demonstrates how to find the value of the most expensive item:

► Input

```
SELECT prod_price
FROM products
ORDER BY prod_price DESC
LIMIT 1;
```

► Output

```
+------------+
| prod_price |
+------------+
|      55.00 |
+------------+
```

► Analysis

`prod_price DESC` ensures that rows are retrieved from most to least expensive, and `LIMIT 1` tells MySQL to return just one row.

Caution

Position of `ORDER BY` Clause When specifying an `ORDER BY` clause, be sure that it is after the `FROM` clause. If `LIMIT` is used, it must come *after* `ORDER BY`. If you use clauses out of order, you will get an error message.

Tip

A Better Way to Find the Largest Value Here we used `ORDER BY` and `LIMIT` to find the most expensive product in a table. In Chapter 12, "Summarizing Data," we'll look at a far more efficient way of finding largest (and smallest and average and other) values.

Summary

In this chapter, you learned how to sort retrieved data by using the `SELECT` statement's `ORDER BY` clause. This clause, which must be the last in the `SELECT` statement, can be used to sort data on one or more columns, as needed.

Challenges

1. Write a SQL statement to retrieve all customer names (`cust_names`) from the `Customers` table and display the results sorted from Z to A.
2. Write a SQL statement to retrieve customer ID (`cust_id`) and order number (`order_num`) from the `Orders` table and sort the results first by customer ID and then by order date, in reverse chronological order.
3. Our fictitious store obviously prefers to sell more expensive items—and lots of them. Write a SQL statement to display the quantity and price (`item_price`) from the `OrderItems` table, sorted with the highest quantity and highest price first.
4. What is wrong with the following SQL statement? (Try to figure it out without running it.)

```
SELECT vend_name,
FROM Vendors
ORDER vend_name DESC;
```

6

Filtering Data

In this chapter, you will learn how to use the `SELECT` statement's `WHERE` clause to specify search conditions.

Using the WHERE Clause

Database tables usually contain large amounts of data, and you seldom need to retrieve all the rows in a table. More often than not, you'll want to extract a subset of a table's data as needed for specific operations or reports. Retrieving just the data you want involves specifying *search criteria*, also known as *filter conditions*.

Within a `SELECT` statement, you filter data by specifying search criteria in the `WHERE` clause. You include the `WHERE` clause right after the table name (the `FROM` clause) as follows:

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE prod_price = 2.50;
```

► Analysis

This statement retrieves two columns from the `products` table, but instead of returning all rows, it returns only rows with the `prod_price` value `2.50`, shown here:

► Output

```
+--------------+--------------- +
| prod_name    | prod_price     |
+--------------+--------------- +
| Carrots      |        2.50    |
| TNT (1 stick) |        2.50    |
+--------------+-----------     +
```

This example uses a simple equality test: It checks to see if a column has a specified value, and it filters the data accordingly. But SQL enables you to do more than just test for equality.

Tip

SQL Versus Application Filtering You can filter data at the application level. To do this, you use the `SQL SELECT` statement to retrieve more data than is actually required for the client application, and the client code loops through the returned data to extract just the needed rows.

As a rule, this practice is strongly discouraged. Databases are optimized to perform filtering quickly and efficiently. Making the client application (or development language) do the database's job dramatically impacts application performance and creates applications that cannot scale properly. In addition, if data is filtered at the client, the server has to send unneeded data across the network connections, resulting in wasted network bandwidth resources.

Caution

`WHERE` Clause Position When using both `ORDER BY` and `WHERE` clauses, make sure `ORDER BY` comes after `WHERE`; otherwise, you will get an error. (See Chapter 5, "Sorting Retrieved Data," for more information on using `ORDER BY`.)

WHERE Clause Operators

New Term

Operator A special keyword used to join or change clauses within a `WHERE` clause. Also known as a *logical operator*.

The first `WHERE` clause we looked at tests for equality to determine whether a column contains a specific value. MySQL supports a whole range of conditional operators, some of which are listed in Table 6.1.

Table 6.1 `WHERE` Clause Operators

Operator	Description
=	Equality
<>	Nonequality
!=	Nonequality
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
BETWEEN	Between two specified values

Checking Against a Single Value

We have already seen an example of testing for equality. Here's one more:

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE prod_name = 'fuses';
```

► Output

```
+-----------+-------------- +
| prod_name | prod_price    |
+-----------+-------------- +
| Fuses       |        3.42 |
+-----------+-------------- +
```

► Analysis

`WHERE prod_name = 'fuses'` returns a single row with the value `Fuses`. By default, MySQL is not case-sensitive when performing matches, and so `fuses` and `Fuses` match.

Now let's look at a few examples that demonstrate the use of other operators.

This example lists all products that cost less than `10`:

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE prod_price < 10;
```

► Output

```
+---------------+---------------  +
| prod_name     | prod_price      |
+---------------+---------------  +
| .5 ton anvil   |          5.99  |
| 1 ton anvil    |          9.99  |
| Carrots        |          2.50  |
| Fuses          |          3.42  |
| Oil can        |          8.99  |
| Sling          |          4.49  |
| TNT (1 stick)  |          2.50  |
+--------------- +--------------- +
```

This statement retrieves all products that cost `10` or less (resulting in two additional matches):

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE prod_price <= 10;
```

Output

```
+---------------+------------ +
| prod_name       | prod_price |
+---------------+------------ +
| .5 ton anvil   |       5.99  |
| 1 ton anvil    |       9.99  |
| Bird seed      |      10.00  |
| Carrots        |       2.50  |
| Fuses          |       3.42  |
| Oil can        |       8.99  |
| Sling          |       4.49  |
| TNT (1 stick)  |       2.50  |
| TNT (5 sticks) |      10.00  |
+---------------+-------------+
```

Checking for Nonmatches

This example lists all products not made by vendor 1003:

Input

```
SELECT vend_id, prod_name
FROM products
WHERE vend_id <> 1003;
```

Output

```
+---------+---------------- +
| vend_id | prod_name       |
+---------+---------------- +
|     1001 | .5 ton anvil   |
|     1001 | 1 ton anvil    |
|     1001 | 2 ton anvil    |
|     1002 | Fuses          |
|     1005 | JetPack 1000   |
|     1005 | JetPack 2000   |
|     1002 | Oil can        |
+---------+---------------- +
```

Tip

When to Use Quotes If you look closely at the conditions used in the `WHERE` clauses in the preceding examples, you will notice that some values are enclosed within single quotes (such as `'fuses'`), and others are not. The single quotes are used to delimit strings. If you are comparing a value against a column that is a string datatype, the delimiting quotes are required. Quotes are not used to delimit values used with numeric columns.

The following example is the same as the previous one except that this one uses the != operator instead of <>:

Input

```
SELECT vend_id, prod_name
FROM products
WHERE vend_id != 1003;
```

Tip

Use <> and != Interchangeably The operator != means *not equal*, so vend_id != 1003 means "match all vendors where vend_id is *not* 1003." The operator <> means *less than or greater than*, so vend_id <> 1003 means "match all vendors where vend_id is less than or greater than 1003 (but not the same)." As you can see, these two operators effectively do the same thing; you can use whichever one you prefer.

Checking for a Range of Values

To check for a range of values, you can use the BETWEEN operator. Its syntax is a little different from that of other WHERE clause operators because it requires two values: the beginning of the range and the end of the range. The BETWEEN operator can be used, for example, to check for all products that cost between 5 and 10 or for all dates that fall between specified start and end dates.

The following example demonstrates the use of the BETWEEN operator to retrieve all products with a price between 5 and 10:

Input

```
SELECT prod_name, prod_price
FROM products
WHERE prod_price BETWEEN 5 AND 10;
```

Output

```
+----------------+-------------- +
| prod_name      | prod_price   |
+----------------+-------------- +
| .5 ton anvil   |        5.99  |
| 1 ton anvil    |        9.99  |
| Bird seed      |       10.00  |
| Oil can        |        8.99  |
| TNT (5 sticks) |       10.00  |
+----------------+-------------- +
```

Analysis

As you can see in this example, when BETWEEN is used, two values must be specified—the low end and the high end of the desired range. The two values must be separated by the AND keyword. BETWEEN matches all the values in the range, including the specified range start and end values.

Checking for No Value

When a table is created, the table designer can specify whether individual columns can contain no value. When a column contains no value, it is said to contain a NULL value.

New Term

NULL No value, as opposed to a field containing 0, or an empty string, or just spaces.

The SELECT statement has a special WHERE clause that can be used to check for columns with NULL values: the IS NULL clause. It looks like this:

► Input

```
SELECT prod_name
FROM products
WHERE prod_price IS NULL;
```

► Analysis

This statement returns a list of all products that have no price (that is, an empty prod_price field, not a price of 0), and because there are none, no data is returned.

The customers table, however, does contain columns with NULL values. The cust_email column contains NULL if a customer has no email address on file, and IS NULL can be used to identify these customers.

► Input

```
SELECT cust_name
FROM customers
WHERE cust_email IS NULL;
```

► Output

```
+------------ +
| cust_name   |
+------------ +
| Mouse House |
| E Fudd      |
+------------ +
```

Caution

NULL and Nonmatches You might expect that when you filter to select all rows that do not have a particular value, rows with a NULL will be returned. But they will not. Because of the special meaning of NULL, the database does not know whether they match, and so they are not returned when filtering for matches or when filtering for nonmatches.

When filtering data, make sure to verify that the rows with NULL in the filtered column are really present in the returned data.

Summary

In this chapter, you learned how to filter returned data by using the SELECT statement's WHERE clause. You learned how to test for equality, nonequality, greater than and less than, value ranges, and NULL values.

Challenges

1. Write a SQL statement to retrieve the product ID (prod_id) and name (prod_name) from the Products table and return only products with a price of 9.49.
2. Write a SQL statement to retrieve the product ID (prod_id) and name (prod_name) from the Products table and return only products with a price of 9 or more.
3. Now you'll test what you learned in Chapter 5 and this chapter. Write a SQL statement that retrieves the unique list of order numbers (order_num) from the OrderItems table for orders that contain 100 or more of any item.
4. Write a SQL statement that returns the product name (prod_name) and price (prod_price) from Products for all products priced between 3 and 6. Oh, and sort the results by price. (There are multiple solutions to this one, and we'll revisit it in the next chapter, but you can solve it using what you've learned thus far.)

This page intentionally left blank

7

Advanced Data Filtering

In this chapter, you'll learn how to combine WHERE clauses to create powerful and sophisticated search conditions. You'll also learn how to use the NOT and IN operators.

Combining WHERE Clauses

All the WHERE clauses introduced in Chapter 6, "Filtering Data," filter data using a single criterion. For a greater degree of filter control, MySQL allows you to specify multiple WHERE clauses. These clauses may be used in two ways: as AND clauses or as OR clauses.

Using the AND Operator

To filter by more than one column, you use the AND operator to append conditions to your WHERE clause. The following code demonstrates this:

► Input

```
SELECT prod_id, prod_price, prod_name
FROM products
WHERE vend_id = 1003 AND prod_price <= 10;
```

► Analysis

This SQL statement retrieves the product name and price for every product made by vendor 1003 as long as the price is 10 or less. The WHERE clause in this SELECT statement is made up of two conditions, and the keyword AND is used to join them. AND instructs MySQL to return only rows that meet all the conditions specified. If a product is made by vendor 1003 but costs more than 10, it is not retrieved. Similarly, products that cost less than 10 that are made by a vendor other than the one specified are not retrieved. The output generated by this SQL statement is as follows:

► Output

```
+---------+------------+----------------+
| prod_id | prod_price | prod_name      |
+---------+------------+----------------+
| FB      |      10.00 | Bird seed      |
| FC      |       2.50 | Carrots        |
| SLING   |       4.49 | Sling          |
```

```
| TNT1    |       2.50 | TNT (1 stick)  |
| TNT2    |      10.00 | TNT (5 sticks) |
+---------+------------+----------------+
```

New Term

AND A keyword used in a WHERE clause to specify that only rows matching all the specified conditions should be retrieved.

This example contains a single AND clause and is thus made up of two filter conditions. Additional filter conditions could be used as well, each separated by an AND keyword.

Using the OR Operator

The OR operator is exactly the opposite of AND. The OR operator instructs MySQL to retrieve rows that match either condition.

Look at the following SELECT statement:

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE vend_id = 1002 OR vend_id = 1003;
```

► Analysis

This SQL statement retrieves the product name and price for any products made by either of the two specified vendors. The OR operator tells MySQL to match either condition—not both. If an AND operator were used here, no data would be returned. (It would create a WHERE clause that can never be matched.) The output generated by this SQL statement is as follows:

► Output

```
+---------------+------------+
| prod_name     | prod_price |
+---------------+------------+
| Detonator     |      13.00 |
| Bird seed     |      10.00 |
| Carrots       |       2.50 |
| Fuses         |       3.42 |
| Oil can       |       8.99 |
| Safe          |      50.00 |
| Sling         |       4.49 |
| TNT (1 stick) |       2.50 |
| TNT (5 sticks) |      10.00 |
+---------------+------------+
```

New Term

OR A keyword used in a WHERE clause to specify that any rows matching either of the specified conditions should be retrieved.

Understanding the Order of Evaluation

WHERE clauses can contain any number of AND and OR operators. By combining the two operators, you can perform sophisticated and complex filtering.

But combining AND and OR operators presents an interesting problem. For example, say that you need a list of all products that cost 10 or more and that were made by vendors 1002 and 1003. The following SELECT statement uses a combination of AND and OR operators to build a WHERE clause for this search:

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE vend_id = 1002
   OR vend_id = 1003
   AND prod_price >= 10;
```

► Output

```
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| Detonator      |      13.00 |
| Bird seed      |      10.00 |
| Fuses          |       3.42 |
| Oil can        |       8.99 |
| Safe           |      50.00 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
```

► Analysis

Look at these results. Two of the rows returned have prices less than 10—so, obviously, the rows were not filtered as intended. Why did this happen? Because of the order of evaluation. SQL (like most other languages) processes AND operators before OR operators. When SQL sees the preceding WHERE clause, it reads "products made by vendor 1002 regardless of price and any products costing 10 or more made by vendor 1003." In other words, because AND ranks higher in the order of evaluation, the wrong operators are joined together here.

The solution to this problem is to use parentheses to explicitly group related operators. Take a look at the following SELECT statement and its output:

► Input

```
SELECT prod_name, prod_price
FROM products
```

```
WHERE (vend_id = 1002 OR vend_id = 1003)
   AND prod_price >= 10;
```

Output

```
+---------------+------------+
| prod_name     | prod_price |
+---------------+------------+
| Detonator     |      13.00 |
| Bird seed     |      10.00 |
| Safe          |      50.00 |
| TNT (5 sticks) |      10.00 |
+---------------+------------+
```

Analysis

The only difference between this `SELECT` statement and the earlier one is that, in this statement, the first two `WHERE` clause conditions are enclosed within parentheses. Because parentheses have a higher order of evaluation than either `AND` or `OR` operators, MySQL first filters the `OR` condition within those parentheses. The SQL statement then looks for any products made by either vendor 1002 or vendor 1003 that cost 10 or greater, which is exactly what we want.

Tip

Using Parentheses in `WHERE` Clauses Whenever you write `WHERE` clauses that use both `AND` and `OR` operators, use parentheses to explicitly group the operators. Don't ever rely on the default evaluation order, even if it is exactly what you want. There is no downside to using parentheses, and you are always better off eliminating any ambiguity.

Using the `IN` Operator

Parentheses have another very different use in `WHERE` clauses. The `IN` operator is used to specify a range of conditions, any of which can be matched. `IN` takes a comma-delimited list of valid values, all enclosed within parentheses. The following example demonstrates this:

Input

```
SELECT prod_name, prod_price
FROM products
WHERE vend_id IN (1002,1003)
ORDER BY prod_name;
```

Output

```
+---------------+------------+
| prod_name     | prod_price |
+---------------+------------+
| Bird seed     |      10.00 |
```

```
| Carrots        |       2.50 |
| Detonator      |      13.00 |
| Fuses          |       3.42 |
| Oil can        |       8.99 |
| Safe           |      50.00 |
| Sling          |       4.49 |
| TNT (1 stick)  |       2.50 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
```

► Analysis

The SELECT statement retrieves all products made by vendors 1002 and 1003. The IN operator is followed by a comma-delimited list of valid values, and the entire list must be enclosed within parentheses.

You might be thinking that the IN operator accomplishes the same goal as OR—and that is correct. The following SQL statement accomplishes exactly the same thing as the previous example:

► Input

```
SELECT prod_name, prod_price
FROM products
WHERE vend_id  = 1002 OR vend_id = 1003
ORDER BY prod_name;
```

► Output

```
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| Bird seed      |      10.00 |
| Carrots        |       2.50 |
| Detonator      |      13.00 |
| Fuses          |       3.42 |
| Oil can        |       8.99 |
| Safe           |      50.00 |
| Sling          |       4.49 |
| TNT (1 stick)  |       2.50 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
```

There are several advantages to using the IN operator:

- When you are working with long lists of valid options, the IN operator syntax is far cleaner and easier to read.
- The order of evaluation is easier to manage when IN is used (as fewer operators are used).
- IN operators almost always execute more quickly than lists of OR operators.
- The biggest advantage of IN is that the IN operator can contain another SELECT statement, enabling you to build highly dynamic WHERE clauses. You'll look at this in detail in Chapter 14, "Working with Subqueries."

New Term

IN A keyword used in a WHERE clause to specify a list of values to be matched using an OR comparison.

Using the NOT Operator

The WHERE clause's NOT operator has one function and one function only: It negates whatever condition comes next.

New Term

NOT A keyword used in a WHERE clause to negate a condition.

The following example demonstrates the use of NOT. To list the products made by all vendors except vendor DLL01, you can use the following:

Input

```
SELECT prod_name
FROM Products
WHERE NOT vend_id = 'DLL01'
ORDER BY prod_name;
```

Output

```
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Bird seed      |
| Carrots        |
| Detonator      |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
```

Analysis

The NOT here negates the condition that follows it. So instead of matching vend_id to DLL01, MySQL matches vend_id to anything that is not DLL01.

This example could also be written using != instead of NOT (as you saw in Chapter 6):

Input

```
SELECT prod_name
FROM Products
WHERE vend_id  != 'DLL01'
ORDER BY prod_name;
```

Actually, it could also be written as:

Input

```
SELECT prod_name
FROM Products
WHERE vend_id  <> 'DLL01'
ORDER BY prod_name;
```

Analysis

Why use NOT? Well, for simple WHERE clauses such as the ones shown here, there really is no advantage to using NOT.

But NOT is extremely useful in more complex clauses. For example, using NOT in conjunction with an IN operator makes it simple to find all rows that do not match a list of criteria. The following example demonstrates this. To list the products made by all vendors except vendors 1002 and 1003, you can use the following:

Input

```
SELECT prod_name, prod_price
FROM products
WHERE vend_id NOT IN (1002,1003)
ORDER BY prod_name;
```

Output

```
+--------------+------------+
| prod_name    | prod_price |
+--------------+------------+
| .5 ton anvil |       5.99 |
| 1 ton anvil  |       9.99 |
| 2 ton anvil  |      14.99 |
| JetPack 1000 |      35.00 |
| JetPack 2000 |      55.00 |
+--------------+------------+
```

Analysis

The NOT here negates the condition that follows it. So instead of matching vend_id to 1002 or 1003, MySQL matches vend_id to anything that is not 1002 or 1003.

So why use NOT? Well, for simple WHERE clauses, there really is no advantage to using NOT. NOT is useful in more complex clauses. For example, using NOT in conjunction with an IN operator makes it easy to find all rows that do not match a list of criteria.

Tip

There Is Often More Than One Solution As you've seen here, there is frequently more than one way to write a SQL statement. When you're working with large sets of data, there may be performance differences, such as one statement being faster than another. But for smaller data sets, the syntax used is usually a matter of personal preference.

Note

`NOT` in MySQL MySQL supports the use of `NOT` to negate `IN`, `BETWEEN`, and `EXISTS` clauses. This is quite different from most other DBMSs, which allow `NOT` to be used to negate any conditions.

Summary

This chapter picked up where the previous chapter left off and taught you how to combine `WHERE` clauses with the `AND` and `OR` operators. You also learned how to explicitly manage the order of evaluation and how to use the `IN` and `NOT` operators.

Challenges

1. Write a SQL statement to retrieve the vendor name (`vend_name`) from the `Vendors` table and return only vendors in California. (This requires filtering by both country [`USA`] and state [`CA`]; after all, there could be a California outside of the United States.) Here's a hint: The filter requires matching strings.
2. Write a SQL statement to find all orders where at least 100 of items BR01, BR02, or BR03 were ordered. You'll want to return the order number (`order_num`), product ID (`prod_id`), and quantity for the `OrderItems` table and filter by both the product ID and quantity. Here's a hint: Depending on how you write your filter, you may need to pay special attention to the order of evaluation.
3. Now let's revisit a challenge from the previous lesson. Write a SQL statement that returns the product name (`prod_name`) and price (`prod_price`) from `Products` for all products priced between 3 and 6. Use the `AND` operator and sort the results by price.
4. What is wrong with the following SQL statement? Try to figure it out without running it.

```
SELECT vend_name
FROM Vendors
ORDER BY vend_name
WHERE vend_country = 'USA' AND vend_state = 'CA';
```

8

Using Wildcard Filtering

In this chapter, you'll learn what wildcards are, how they are used, and how to perform wildcard searches by using the `LIKE` operator for sophisticated filtering of retrieved data.

Using the `LIKE` Operator

All the operators we have studied so far filter against known values. Whether matching one or more values, testing for greater-than or less-than known values, or checking a range of values, the common denominator is that the values used in the filtering are known. But filtering data that way does not always work. For example, how could you search for all products that contained the text *anvil* within the product name? You cannot do that with simple comparison operators; it's a job for wildcard searching.

You can use wildcards to create search patterns that can be compared against your data. For example, say that to find all products that contain the words *anvil*, you could construct a wildcard search pattern that finds the text *anvil* anywhere within a product name.

New Term

Wildcard A special character used to match part of a value.

New Term

Search pattern A search condition made up of literal text, wildcard characters, or any combination of the two.

The wildcards themselves are actually characters that have special meanings within SQL `WHERE` clauses, and SQL supports several wildcard types.

To use wildcards in search clauses, you use the `LIKE` operator. `LIKE` instructs MySQL to compare the search pattern that follows by using a wildcard match rather than a straight equality match.

Note

Predicates When is an operator not an operator? When it is a *predicate*. Technically, `LIKE` is a predicate, not an operator. Be aware of this term in case you run across it in the MySQL documentation.

The Percent Sign (%) Wildcard

The most frequently used wildcard is the percent sign (`%`). Within a search string, `%` means "match any number of occurrences of any character." For example, to find all products that start with the word `jet`, you can issue the following `SELECT` statement:

Input

```
SELECT prod_id, prod_name
FROM products
WHERE prod_name LIKE 'jet%';
```

Output

```
+---------+--------------+
| prod_id | prod_name    |
+---------+--------------+
| JP1000  | JetPack 1000 |
| JP2000  | JetPack 2000 |
+---------+--------------+
```

Analysis

This example uses the search pattern `'jet%'`. When this clause is evaluated, any value that starts with `jet` is retrieved. The `%` tells MySQL to accept any characters after the word `jet`, regardless of how many characters there are.

Note

Case-Sensitivity Depending on how MySQL is configured, searches might be case-sensitive, in which case `'jet%'` would not match `JetPack 1000`.

Wildcards can be used anywhere within the search pattern, and multiple wildcards can be used. The following example uses two wildcards, one at either end of the pattern:

Input

```
SELECT prod_id, prod_name
FROM products
WHERE prod_name LIKE '%anvil%';
```

Output

```
+---------+--------------+
| prod_id | prod_name    |
+---------+--------------+
| ANV01   | .5 ton anvil |
| ANV02   | 1 ton anvil  |
| ANV03   | 2 ton anvil  |
+---------+--------------+
```

Analysis

The search pattern `'%anvil%'` means "match any value that contains the text `anvil` anywhere within it, regardless of any characters before or after that text."

You can also use wildcards in the middle of a search pattern, although doing that is rarely useful. The following example finds all products that begin with an `s` and end with an `e`:

Input

```
SELECT prod_name
FROM products
WHERE prod_name LIKE 's%e';
```

It is important to note that, in addition to matching one or more characters, `%` also matches zero characters. `%` represents zero, one, or more characters at the specified location in the search pattern.

Note

Watch for Trailing Spaces Trailing spaces can interfere with wildcard matching. For example, if any of the anvils were saved with one or more spaces after the word *anvil*, the clause `WHERE prod_name LIKE '%anvil'` would not match them as there would be additional characters after the final `l`. One simple solution to this problem is to always append a final `%` to the search pattern. A better solution is to trim the spaces by using functions, as discussed in Chapter 11, "Using Data Manipulation Functions."

Caution

Watch for `NULL` Although it might seem that the `%` wildcard matches anything, there is one exception: `NULL`. Not even the clause `WHERE prod_name LIKE '%'` will match a row with the value `NULL` as the product name.

The Underscore (_) Wildcard

Another useful wildcard is the underscore (_). The underscore is used just like `%`, but instead of matching multiple characters, the underscore matches just a single character.

Take a look at this example:

Input

```
SELECT prod_id, prod_name
FROM products
WHERE prod_name LIKE '_ ton anvil';
```

Output

```
+---------+-------------+
| prod_id | prod_name   |
+---------+-------------+
| ANV02   | 1 ton anvil |
| ANV03   | 2 ton anvil |
+---------+-------------+
```

Analysis

The search pattern used in this `WHERE` clause specifies a wildcard followed by literal text. The results shown are the only rows that match the search pattern: The underscore matches `1` in the first row and `2` in the second row. It does not match the `.5 ton anvil` product because the search pattern matches a single character, not two characters. By contrast, the following `SELECT` statement uses the `%` wildcard and returns three matching products:

Input

```
SELECT prod_id, prod_name
FROM products
WHERE prod_name LIKE '% ton anvil';
```

Output

```
+---------+--------------+
| prod_id | prod_name    |
+---------+--------------+
| ANV01   | .5 ton anvil |
| ANV02   | 1 ton anvil  |
| ANV03   | 2 ton anvil  |
+---------+--------------+
```

Analysis

Unlike `%`, which can match zero characters, _ always matches exactly one character.

Tips for Using Wildcards

As you can see, MySQL's wildcards are extremely powerful. But that power comes with a price: Wildcard searches typically take far longer to process than any of the search types discussed to this point. Here are some tips to keep in mind when using wildcards:

- Don't overuse wildcards. If another search operator will do, use it instead.
- When you use wildcards, try to not use them at the beginning of a search pattern unless absolutely necessary. Search patterns that begin with wildcards are the slowest to process.
- Pay careful attention to the placement of the wildcard symbols. If they are misplaced, the data returned might not be what you intended.

Despite their drawbacks, wildcards are important and useful search tools, and you will use them frequently.

Summary

In this chapter, you learned what SQL wildcards are and how to use them in your `WHERE` clauses. You also learned that you should use wildcards carefully and only where necessary.

Challenges

1. Write a SQL statement to retrieve the product name (`prod_name`) and description (`prod_desc`) from the `Products` table and return only products that have the word `toy` in the description.
2. Now let's flip things around. Write a SQL statement to retrieve the product name (`prod_name`) and description (`prod_desc`) from the `Products` table and return only products that don't have the word `toy` in the description. This time, sort the results by product name.
3. Write a SQL statement to retrieve the product name (`prod_name`) and description (`prod_desc`) from the `Products` table and return only products that have both the words `toy` and `carrots` in the description. There are a couple of ways to do this, but for this challenge, use `AND` and two `LIKE` comparisons.
4. This one is a little trickier. I didn't show you this syntax specifically, but see whether you can figure it out anyway, based on what you have learned thus far. Write a SQL statement to retrieve the product name (`prod_name`) and description (`prod_desc`) from the `Products` table and return only products where both the words `toy` and `carrots` appear in the description, in that order (that is, the word `toy` before the word `carrots`). Here's a hint: You'll only need one `LIKE` with three `%` symbols to do this.

This page intentionally left blank

9

Searching Using Regular Expressions

In this chapter, you'll learn how to use regular expressions within MySQL `WHERE` clauses for greater control over data filtering.

Understanding Regular Expressions

The filtering examples in the previous two chapters enabled you to locate data using matches, comparisons, and wildcard operators. For basic filtering (and even some not-so-basic filtering), this might be enough. But as the complexity of filtering conditions grows, so does the complexity of the `WHERE` clauses themselves—and this is where regular expressions become useful.

A regular expression is a special string (that is, a set of characters) that is used to match text. If you need to extract phone numbers from a text file, for example, you might use a regular expression. If you need to locate all files with digits in the middle of their names, you might use a regular expression. If you want to find all repeated words in a block of text, you might use a regular expression. And if you want to replace all URLs on a page with actual HTML links to those same URLs, you might use a regular expression (or two).

Regular expressions are supported in all sorts of programming languages, text editors, operating systems, and more. Savvy programmers and network managers have long regarded regular expressions as a vital component of their technical toolboxes.

Regular expressions are created using the regular expression language, a specialized language designed to do everything just discussed and much more. Like any other language, regular expression language has special syntax and instructions that you must learn.

Note

To Learn More Full coverage of regular expressions is beyond the scope of this chapter. While the basics are covered here, for a more thorough introduction to regular expressions, you might want to obtain a copy of my book *Learning Regular Expressions* (ISBN 9780134757063).

Using MySQL Regular Expressions

So what do regular expressions have to do with MySQL? As already explained, regular expressions are used to match text by comparing a pattern (the regular expression) with a string of text. MySQL provides rudimentary support for regular expressions with `WHERE` clauses, allowing you to specify regular expressions that are used to filter data retrieved by using `SELECT`.

Note

Just a Subset of the Regular Expression Language If you are already familiar with regular expressions, take note. MySQL supports only a small subset of what most regular expression implementations support, and this chapter covers most of what is supported.

This will all become much clearer with some examples.

Basic Character Matching

We'll start with a very simple example. The following statement retrieves all rows where the column `prod_name` contains the text `1000`:

► Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '1000'
ORDER BY prod_name;
```

► Output

```
+--------------- +
| prod_name      |
+--------------- +
| JetPack 1000   |
+--------------- +
```

► Analysis

This statement looks much like the ones that used `LIKE` in Chapter 8, "Using Wildcard Filtering," except that the keyword `LIKE` has been replaced with `REGEXP`. This tells MySQL that what follows is to be treated as a regular expression (one that just matches the literal text `1000`).

So, why bother using a regular expression? Well, in this example, regular expressions really add no value (and probably hurt performance), but consider this next example:

► Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '.000'
ORDER BY prod_name;
```

Output

```
+--------------- +
| prod_name      |
+--------------- +
| JetPack 1000   |
| JetPack 2000   |
+--------------- +
```

Analysis

Here the regular expression `.000` is used. The period (.) is a special character in regular expression language. It means "match any single character," and so both `1000` and `2000` match and are returned.

Of course, this particular example could also be accomplished by using `LIKE` and wildcards (refer to Chapter 8).

Note

`LIKE` Versus `REGEXP` There is one very important difference between `LIKE` and `REGEXP`. Look at these two statements:

```
SELECT prod_name
FROM products
WHERE prod_name LIKE '1000'
ORDER BY prod_name;
```

and:

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '1000'
ORDER BY prod_name;
```

If you were to try them both, you'd discover that the first one returns no data, and the second one returns one row. Why is this?

As you saw in Chapter 8, `LIKE` matches an entire column. If the text to be matched exists in the middle of a column value, `LIKE` will not find it and will not return the row (unless wildcard characters are used). `REGEXP`, on the other hand, looks for matches within column values, and so if the text to be matched exists in the middle of a column value, `REGEXP` finds it and returns the row. This is a very important distinction.

So can `REGEXP` be used to match entire column values (so that it functions like `LIKE`)? Actually, yes, if you also use the `^` and `$` anchors, as explained later in this chapter.

Tip

Matches Are Not Case-Sensitive Regular expression matching in MySQL is not case-sensitive; that is, either case will be matched. To force case-sensitivity, you can use the `BINARY` keyword, as in this example:

```
WHERE prod_name REGEXP BINARY 'JetPack .000'
```

Performing OR Matches

To search for one of two strings (either one or the other), you use the pipe character (`|`), as shown here:

Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '1000|2000'
ORDER BY prod_name;
```

Output

```
+--------------- +
| prod_name      |
+--------------- +
| JetPack 1000   |
| JetPack 2000   |
+--------------- +
```

Analysis

Here the regular expression `1000|2000` is used. `|` is the regular expression `OR` operator. It means "match one or the other," and so both `1000` and `2000` match and are returned.

Using `|` is functionally similar to using `OR` statements in `SELECT` statements, with multiple `OR` conditions being consolidated into a single regular expression.

Tip

More Than Two `OR` Conditions You can specify more than two `OR` conditions. For example, you can use `'1000|2000|3000'` to match `1000` or `2000` or `3000`.

Matching One of Several Characters

The character `.` matches any single character. But what if you want to match only specific characters? You can do this by specifying a set of characters enclosed within `[` and `]`, as shown here:

Input

```
SELECT prod_name
FROM products
```

```
WHERE prod_name REGEXP '[123] Ton'
ORDER BY prod_name;
```

► Output

```
+-------------- +
| prod_name     |
+-------------- +
| 1 ton anvil   |
| 2 ton anvil   |
+-------------- +
```

► Analysis

Here the regular expression `[123] Ton` is used. `[123]` defines a set of characters, and it says to match `1` or `2` or `3`. In this case, both `1 ton` and `2 ton` match and are returned (and there is no `3 ton`).

As you have just seen, `[]` is another form of an `OR` statement. In fact, the regular expression `[123] Ton` is shorthand for `[1|2|3] Ton`, which would also work. But the `[]` characters are needed to define what the `OR` statement is looking for. To better understand this, look at this example:

► Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '1|2|3 Ton'
ORDER BY prod_name;
```

► Output

```
+---------------- +
| prod_name       |
+---------------- +
| 1 ton anvil     |
| 2 ton anvil     |
| JetPack 1000    |
| JetPack 2000    |
| TNT (1 stick)   |
+---------------- +
```

► Analysis

Well, this doesn't work. The two required rows are retrieved, but so are three others. This happens because MySQL assumes that you mean to match `1 ton` or `2 ton` or `3 ton`. The `|` character applies to the entire string unless it is enclosed with a set.

Sets of characters can also be negated. That is, you can tell MySQL to match anything *except* the specified characters. To negate a character set, place a `^` at the start of the set. So, whereas `[123]` matches characters `1`, `2`, or `3`, `[^123]` matches anything except those characters.

Matching Ranges

You can use a set to define one or more characters to be matched. For example, the following set matches digits 0 through 9:

```
[0123456789]
```

To simplify this type of set, you can use - to define a range. The following is functionally identical to `[0123456789]`:

```
[0-9]
```

Ranges are not limited to complete sets, so `[1-3]` and `[6-9]` are valid ranges, too. In addition, ranges need not be numeric; for example, `[a-z]` will match any alphabetical character.

Here is an example:

► Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '[1-5] Ton'
ORDER BY prod_name;
```

► Output

```
+--------------- +
| prod_name      |
+--------------- +
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
+--------------- +
```

► Analysis

Here the regular expression `[1-5] Ton` is used. `[1-5]` defines a range, and this expression means "match 1 through 5." In this case, three matches are returned. `.5 ton` is returned because `5 ton` matches (without the `.` character).

Matching Special Characters

The regular expression language is made up of special characters that have specific meanings. You've already seen `.`, `[]`, `|`, and `-`, and there are others, too. So if you need to match those characters, how can you do it? For example, if you want to find values that contain the `.` character, how can you search for it? Look at this example:

► Input

```
SELECT vend_name
FROM vendors
WHERE vend_name REGEXP '.'
ORDER BY vend_name;
```

Output

```
+---------------- +
| vend_name       |
+---------------- +
| ACME            |
| Anvils R Us     |
| Furball Inc.    |
| Jet Set         |
| Jouets Et Ours  |
| LT Supplies     |
+---------------- +
```

Analysis

This did not work. . matches any character, and so every row is retrieved.

To match special characters, you need to precede them with \\. So, for example, \\- means "find -," and, as shown in this example, \\. means "find .":

Input

```
SELECT vend_name
FROM vendors
WHERE vend_name REGEXP '\\.'
ORDER BY vend_name;
```

Output

```
+--------------- +
| vend_name      |
+--------------- +
| Furball Inc.   |
+--------------- +
```

Analysis

This works. \\. matches ., and so only a single row is retrieved. This process is known as *escaping*, and all characters that have special significance within regular expressions must be escaped this way—including ., |, [], and all of the other special characters used thus far.

\\ can also be used to refer to metacharacters (that is, characters that have specific meanings), such as the ones listed in Table 9.1.

Table 9.1 White Space Metacharacters

Metacharacter	Description
\\f	Form feed
\\n	Line feed
\\r	Carriage return
\\t	Tab
\\v	Vertical tab

Tip

**To Match ** To match the backslash character itself (\), you need to use `\\\`.

Note

\ or \\? With most regular expression implementations, you use a single backslash to escape special characters in order to use them as literals. MySQL, however, requires two backslashes. (MySQL interprets one of the backslashes, and the regular expression library interprets the other one.)

Matching Character Classes

There are matches that you'll find yourself using frequently: digits, all alphabetical characters, all alphanumerical characters, and so on. To make working with these matches easier, you can use predefined character sets known as *character classes*. Table 9.2 describes the available character classes.

Table 9.2 Character Classes

Class	Description
`[:alnum:]`	Any letter or digit (same as `[a-zA-Z0-9]`)
`[:alpha:]`	Any letter (same as `[a-zA-Z]`)
`[:blank:]`	Space or tab (same as `[\\t ]`)
`[:cntrl:]`	ASCII control characters (ASCII 0 through 31 and 127)
`[:digit:]`	Any digit (same as `[0-9]`)
`[:graph:]`	Any printable character (same as `[:print:]` but excludes space)
`[:lower:]`	Any lowercase letter (same as `[a-z]`)
`[:print:]`	Any printable character
`[:punct:]`	Any character that is in neither `[:alnum:]` nor `[:cntrl:]`
`[:space:]`	Any white space character, including the space (same as `[\\f\\n\\r\\t\\v ]`)
`[:upper:]`	Any uppercase letter (same as `[A-Z]`)
`[:xdigit:]`	Any hexadecimal digit (same as `[a-fA-F0-9]`)

Matching Multiple Instances

All of the regular expressions used thus far attempt to match a single occurrence. If there is a match, the row is retrieved; if there is not a match, nothing is retrieved. But sometimes you'll require greater control over the number of matches. For example, you might want to locate all numbers, regardless of how many digits a number contains, or

you might want to locate a word but also want to be able to accommodate a trailing *s* if one exists, and so on.

These sorts of matches can be accomplished by using the regular expressions repetition metacharacters, listed in Table 9.3.

Table 9.3 Repetition Metacharacters

Metacharacter	Description
*	Zero or more matches
+	One or more matches (equivalent to {1,})
?	Zero or one match (equivalent to {0,1})
{n}	Specified number of matches
{n,}	No fewer than a specified number of matches
{n,m}	Range of matches (m not to exceed 255)

Consider this example:

Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '\\([0-9] sticks?\\)'
ORDER BY prod_name;
```

Output

```
+---------------- +
| prod_name       |
+---------------- +
| TNT (1 stick)   |
| TNT (5 sticks)  |
+---------------- +
```

Analysis

The regular expression `\\([0-9] sticks?\\)` requires some explanation. `\\(` matches `(`, `[0-9]` matches any digit (`1` and `5` in this example), `sticks?` matches `stick` and `sticks` (the `?` after the `s` makes that `s` optional because `?` matches `0` or `1` occurrence of whatever it follows), and `\\)` matches the closing `)`. Without the `?` metacharacter, it would be very difficult to match both `stick` and `sticks`.

Here's an example that tries to match four consecutive digits:

Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '[[:digit:]]{4}'
ORDER BY prod_name;
```

Output

```
+--------------- +
| prod_name      |
+--------------- +
| JetPack 1000   |
| JetPack 2000   |
+--------------- +
```

Analysis

As explained previously, `[:digit:]` matches any digit, and so `[[:digit:]]` is a set of digits. `{4}` requires exactly four occurrences of whatever it follows (in this case, any digit), and so `[[:digit:]]{4}` matches any four consecutive digits.

It is worth noting that when using regular expressions, there is almost always more than one way to write a specific expression. The previous example could also be written as follows:

Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '[0-9][0-9][0-9][0-9]'
ORDER BY prod_name;
```

Anchors

All of the examples thus far have matched text anywhere within a string. To match text at specific locations, you need to use anchor metacharacters, as listed in Table 9.4.

Table 9.4 Anchor Metacharacters

Metacharacter	Description
^	Start of text
$	End of text
`[[:<:]]`	Start of word
`[[:>:]]`	End of word

For example, what if you wanted to find all products that start with a number (including numbers that start with a decimal point)? A simple search for `[0-9\\.]` (or `[[:digit:]\\.]`) would not work because it would find matches anywhere within the text. The solution is to use the `^` anchor, as shown here:

Input

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '^[0-9\\.]'
ORDER BY prod_name;
```

Output

```
+--------------- +
| prod_name      |
+--------------- +
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
+--------------- +
```

Analysis

`^` matches the start of a string. Therefore, `^[0-9\\.]` matches `.` or any digit only if they are the first characters within a string. Without the `^`, four other rows would be retrieved, too (those that have digits in the middle).

Note

The Dual-Purpose `^` `^` has two uses. It can be used within a set (defined using `[` and `]`) to negate that set. Otherwise, it is used to refer to the start of a string.

Note

Making `REGEXP` Behave Like `LIKE` Earlier in this chapter, I mentioned that `LIKE` and `REGEXP` behave differently in that `LIKE` matches an entire string, and `REGEXP` matches substrings, too. When you use anchors, you can make `REGEXP` behave just like `LIKE` by simply starting each expression with `^` and ending it with `$`.

Tip

Simple Regular Expression Testing You can use `SELECT` to test regular expressions without using database tables. `REGEXP` checks always return `0` (for no match) or `1` (for a match). You can use `REGEXP` with literal strings to test expressions and to experiment with them. Here's an example:

```
SELECT 'hello' REGEXP '[0-9]';
```

This example would return `0` because there are no digits in the text `hello`.

Summary

In this chapter, you learned the basics of regular expressions and how to use them in MySQL `SELECT` statements via the `REGEXP` keyword.

Challenges

1. Use regular expressions to return every product whose name ends with a number.
2. In this chapter, you learned how to use `REGEXP` to match text containing digits. Can you figure out how to match only products with no digits in their names? Here's a hint: You can negate an entire match (as you learned in Lesson 7, "Advanced Data Filtering").
3. This final one is a little trickier. Some of the products listed in `products` have names that are made up of more than one word. Use regular expressions to return only the products with names made up of three or more words. Here's a hint: Look for spaces between the words.

10

Creating Calculated Fields

In this chapter, you will learn what calculated fields are, how to create them, and how to use aliases to refer to them from within your application.

Understanding Calculated Fields

Data stored within a database's tables is often not available in the exact format needed by your applications. Here are some examples:

- You need to display a field containing the name of a company along with the company's location, but that information is stored in separated table columns.
- City, state, and zip code are stored in separate columns (as they should be), but your mailing label printing program needs them retrieved as one correctly formatted field.
- Column data is in mixed upper- and lowercase, and your report needs data presented in all uppercase.
- An order items table stores item price and quantity but not the expanded price (that is price multiplied by quantity) for each item. To print invoices, you need that expanded price.
- You need totals, averages, or other calculations based on table data.

In each of these examples, the data stored in the table is not exactly what your application needs. Rather than retrieve the data as it is and then reformat it within your client application or report, you want to be able to retrieve converted, calculated, or reformatted data directly from the database.

This is where calculated fields come in. Unlike all the columns retrieved in the chapters thus far, calculated fields don't actually exist in database tables. Rather, a calculated field is created on-the-fly within a SQL `SELECT` statement.

New Term

Field Essentially the same thing as *column*. These terms are often used interchangeably, although database columns are typically called *columns* and the term *fields* is normally used in conjunction with calculated fields.

It is important to note that only the database knows which columns in a SELECT statement are actual table columns and which are calculated fields. From the perspective of a client (for example, your application), a calculated field's data is returned in the same way as data from any other column.

Tip

Client Versus Server Formatting Many of the conversions and reformatting operations that can be performed within SQL statements can also be performed directly in a client application. However, as a rule, it is far quicker to perform these operations on a database server than it is to perform them within a client because database management systems (DBMSs) are built to perform this type of processing quickly and efficiently.

Concatenating Fields

To demonstrate working with calculated fields, let's start with a simple example: creating a title made up of two columns.

The vendors table contains vendor name and address information. Imagine that you are generating a vendor report and need to list the vendor location as part of the vendor name in the format *name* (*location*).

The report wants a single value, and the data in the table is stored in two columns: vend_name and vend_country. In addition, you need to surround vend_country with parentheses, and those are definitely not stored in the database table. The SELECT statement that returns the vendor names and locations is simple enough, but how would you create the combined value that you need?

The solution is to concatenate the two columns. In MySQL SELECT statements, you can concatenate columns by using the Concat() function.

New Term

Concatenate To join values together (by appending them to each other) to form a single long value.

Tip

MySQL Is Different Most DBMSs use the operators + or || for concatenation; MySQL uses the Concat() function. Keep this in mind when converting SQL statements to MySQL.

Here's an example of using the `Concat()` function:

Input

```
SELECT Concat(vend_name, ' (', vend_country, ')')
FROM vendors
ORDER BY vend_name;
```

Output

```
+-------------------------------------------+
| Concat(vend_name, ' (', vend_country, ')') |
+-------------------------------------------+
| ACME (USA)                                |
| Anvils R Us (USA)                         |
| Furball Inc. (USA)                        |
| Jet Set (England)                         |
| Jouets Et Ours (France)                   |
| LT Supplies (USA)                         |
+-------------------------------------------+
```

Analysis

`Concat()` concatenates strings, appending them to each other to create one bigger string. `Concat()` requires one or more values to be specified, and the values need to be separated by commas. The `SELECT` statements in this example concatenate four elements:

- The name stored in the `vend_name` column
- A string containing a space and an open parenthesis
- The state stored in the `vend_country` column
- A string containing the close parenthesis

As you can see in the output, the `SELECT` statement returns a single column (a calculated field) that contains all four of these elements as one unit.

Back in Chapter 8, "Using Wildcard Filtering," I mentioned the need to trim data in order to remove any trailing spaces. This can be done using the MySQL `RTrim()` function, as shown here:

Input

```
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country), ')')
FROM vendors
ORDER BY vend_name;
```

Analysis

The `RTrim()` function trims all spaces from the right of a value. When you use `RTrim()`, the individual columns are all trimmed properly.

Note

The `Trim()` Functions In addition to `RTrim()` (which, as just seen, trims the right side of a string), MySQL supports the use of `LTrim()` (which trims the left side of a string) and `Trim()` (which trims both the right and left sides).

Tip

Functions Are Not Case-Sensitive As MySQL functions are not case-sensitive, you can concatenate by using `Concat()`, `concat()` and even `CONCAT()`. It comes down to personal preference, and you can feel free to use any style you prefer. But please be consistent: Whatever style you opt for, stick with it. Doing so will make your code much easier to read and maintain in the future.

Using Aliases

The `SELECT` statement that is used to concatenate the address field works well, as you can see in the previous output. But what is the name of the new calculated column? Well, the truth is, it has no name; it is simply a value. Although this can be fine if you are just looking at the results in a SQL query tool, an unnamed column cannot be used within a client application because the client has no way to refer to that column.

To solve this problem, SQL supports column aliases. An `alias` is an alternative name for a field or value. You assign aliases by using the `AS` keyword. Take a look at the following `SELECT` statement:

Input

```
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country), ')')
   AS vend_title
FROM vendors
ORDER BY vend_name;
```

Output

```
+------------------------+
| vend_title             |
+------------------------+
| ACME (USA)             |
| Anvils R Us (USA)      |
| Furball Inc. (USA)     |
| Jet Set (England)      |
| Jouets Et Ours (France)|
| LT Supplies (USA)      |
+------------------------+
```

Analysis

This `SELECT` statement is the same as the one used in the previous code snippet, except that here the calculated field is followed by the text `AS vend_title`. This instructs SQL to create a calculated field named `vend_title` that contains the results of the specified calculation. As you can see in the output, the results are the same as before, but the column is now named `vend_title`, and any client application can refer to this column by name, just as it would to any actual table column.

Tip

Other Uses for Aliases Aliases have other uses, too. Some common uses include renaming a column if the table column name contains illegal characters (for example, spaces) and expanding a column name if the original name is either ambiguous or easily misread.

Note

Derived Columns *Aliases* are also sometimes referred to as *derived columns*. These two terms mean the same thing.

Performing Mathematical Calculations

Another frequent use for calculated fields is performing mathematical calculations on retrieved data. Let's take a look at an example. The `orders` table contains all orders received, and the `orderitems` table contains the individual items within each order. The following SQL statement retrieves all the items in order number 20005:

Input

```
SELECT prod_id, quantity, item_price
FROM orderitems
WHERE order_num = 20005;
```

Output

```
+---------+----------+------------+
| prod_id | quantity | item_price |
+---------+----------+------------+
| ANV01   |       10 |       5.99 |
| ANV02   |        3 |       9.99 |
| TNT2    |        5 |      10.00 |
| FB      |        1 |      10.00 |
+---------+----------+------------+
```

The item_price column contains the per-unit price for each item in an order. To expand the item price (item price multiplied by quantity ordered), you simply use the following:

Input

```
SELECT prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM orderitems
WHERE order_num = 20005;
```

Output

```
+---------+----------+------------+----------------+
| prod_id | quantity | item_price | expanded_price |
+---------+----------+------------+----------------+
| ANV01   |       10 |       5.99 |          59.90 |
| ANV02   |        3 |       9.99 |          29.97 |
| TNT2    |        5 |      10.00 |          50.00 |
| FB      |        1 |      10.00 |          10.00 |
+---------+----------+------------+----------------+
```

Analysis

The expanded_price column shown in the previous output is a calculated field; the calculation is simply quantity*item_price. Once this calculated column is created, the client application can use it just as it would any other column.

MySQL supports the basic mathematical operators listed in Table 10.1. In addition, you can use parentheses to establish the order of precedence. Refer to Chapter 7, "Advanced Data Filtering," for an explanation of precedence.

Table 10.1 MySQL Mathematical Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Tip

How to Test Calculations SELECT provides a great way to test and experiment with functions and calculations. Although SELECT is usually used to retrieve data from a table, the FROM clause may be omitted to simply access and work with expressions. For example, SELECT 3 * 2; would return 6, SELECT Trim(' abc '); would return abc, and SELECT Now() uses the Now() function to return the current date and time. You get the idea, and you can learn more by using SELECT to experiment.

Summary

In this chapter, you learned what calculated fields are and how to create them. You saw examples demonstrating the use of calculated fields for both string concatenation and mathematical operations. In addition, you learned how to create and use aliases so your application can refer to calculated fields.

Challenges

1. A common use for aliases is to rename table column fields in retrieved results (perhaps to match specific reporting or client needs). Write a SQL statement that retrieves `vend_id`, `vend_name`, `vend_address`, and `vend_city` from `Vendors` and rename `vend_name` to `vname`, `vend_city` to `vcity`, and `vend_address` to `vaddress`. Sort the results by vendor name; for this you can use either the original name or the new name.
2. Our example store is running a sale, and all products are 10% off. Write a SQL statement that returns `prod_id`, `prod_price`, and `sale_price` from the `Products` table. `sale_price` is a calculated field that contains the sale price. Here's a hint: You can multiply by `0.9` to get 90% of the original value (and thus the 10% off price).

This page intentionally left blank

11

Using Data Manipulation Functions

In this chapter, you'll learn what functions are, what types of functions MySQL supports, and how to use these functions.

Understanding Functions

Like almost any other computer language, SQL supports the use of functions to manipulate data. Functions are operations that are usually performed on data, typically to facilitate conversion and manipulation.

An example of a function is the `RTrim()` function that we used in the last chapter to trim any spaces from the end of a string.

Note

Functions Are Less Portable Than SQL Code that runs on multiple systems is said to be *portable*. Most SQL statements are relatively portable, and when differences between SQL implementations occur, they are usually not very difficult to deal with. Functions, on the other hand, tend to be far less portable. Just about every major database management system (DBMS) supports functions that other DBMSs don't, and sometimes the differences are significant.

With code portability in mind, many SQL programmers opt not to use any implementation-specific features. Although this is a somewhat noble and idealistic view, it is not always in the best interests of application performance. If you opt not to use implementation-specific functions, you make your application code work harder. It must use other methods to do what the DBMS could have done more efficiently.

If you decide to use implementation-specific functions, make sure you comment your code well, so that at a later date you (or another developer) will know exactly to which SQL implementation you were writing.

Using Functions

Most SQL implementations support the following types of functions:

- **Text:** These functions are used to manipulate strings of text (for example, trimming or padding values and converting values to upper- and lowercase).
- **Numeric:** These functions are used to perform mathematical operations on numeric data (for example, returning absolute numbers and performing algebraic calculations).
- **Date and time:** These functions are used to manipulate date and time values and to extract specific components from these values (for example, returning differences between dates and checking date validity).
- **System:** These functions return information specific to the DBMS being used (for example, returning user login information or checking version specifics).

Text Manipulation Functions

You've already seen an example of text-manipulation functions in the last chapter: You saw how to use the `RTrim()` function to trim white space from the end of a column value. Here is another example, this time using the `Upper()` function:

Input

```
SELECT vend_name, Upper(vend_name) AS vend_name_upcase
FROM vendors
ORDER BY vend_name;
```

Output

```
+---------------+------------------+
| vend_name     | vend_name_upcase |
+---------------+------------------+
| ACME          | ACME             |
| Anvils R Us   | ANVILS R US      |
| Furball Inc.  | FURBALL INC.     |
| Jet Set       | JET SET          |
| Jouets Et Ours | JOUETS ET OURS  |
| LT Supplies   | LT SUPPLIES      |
+---------------+------------------+
```

Analysis

As you can see, `Upper()` converts text to uppercase, and so in this example, each vendor is listed twice: The first time it is listed exactly as it is stored in the `vendors` table, and then it is converted to uppercase in the column `vend_name_upcase`.

Table 11.1 lists some commonly used text-manipulation functions.

Table 11.1 Commonly Used Text-Manipulation Functions

Function	Description
`Left()`	Returns characters from the left of a string
`Length()`	Returns the length of a string
`Locate()`	Finds a substring within a string
`Lower()`	Converts a string to lowercase
`LTrim()`	Trims white space from the left of a string
`Right()`	Returns characters from the right of a string
`RTrim()`	Trims white space from the right of a string
`Soundex()`	Returns a string's SOUNDEX value
`SubString()`	Returns characters from within a string
`Upper()`	Converts a string to uppercase

One item in Table 11.1 requires further explanation. SOUNDEX is an algorithm that converts any string of text into an alphanumeric pattern that describes the phonetic representation of that text. SOUNDEX takes into account similar-sounding characters and syllables to enable strings to be compared based on how they sound rather than how they have been typed. Although SOUNDEX is not a SQL concept, MySQL (like many other DBMSs) offers SOUNDEX support.

Here's an example of using the `Soundex()` function. Customer `Coyote Inc.` is in the `customers` table and has a contact named `Y. Lee`. But what if that is a typo, and the contact is actually supposed to be `Y. Lie`? Obviously, searching by the correct contact name will return no data, as shown here:

Input

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_contact = 'Y. Lie';
```

Output

```
+------------+--------------+
| cust_name  | cust_contact |
+------------+--------------+
```

Now try the same search using the `Soundex()` function to match all contact names that sound similar to `Y. Lie`:

Input

```
SELECT cust_name, cust_contact
FROM customers
WHERE Soundex(cust_contact) = Soundex('Y Lie');
```

Output

```
+-------------+--------------+
| cust_name   | cust_contact |
+-------------+--------------+
| Coyote Inc. | Y Lee        |
+-------------+--------------+
```

Analysis

In this example, the `WHERE` clause uses the `Soundex()` function to convert both the `cust_contact` column value and the search string to their SOUNDEX values. Because `Y. Lee` and `Y. Lie` sound alike, their SOUNDEX values match, and so the `WHERE` clause correctly filters the desired data.

Note

Understanding SOUNDEX In case you are curious about how the `Soundex()` function works its magic, I'll tell you the secret: SOUNDEX is a formula that converts any string of text into a four-character value that represents its sound. The `WHERE` statement used above simply obtains the SOUNDEX value for `Y Lie` and compares it to the SOUNDEX value for every contact in the table:

```
WHERE Soundex(cust_contact) = Soundex('Y Lie');
```

If you're interested, you can use this SQL statement to see every contact and its SOUNDEX value:

```
SELECT cust_contact,
   SOUNDEX(cust_contact) AS cust_contact_soundex
FROM customers
```

Date and Time Manipulation Functions

Dates and times are stored in tables using special datatypes with special internal formats so they can be sorted or filtered quickly and efficiently and in order to save physical storage space.

The format used to store dates and times is usually of no use to your applications, and so date and time functions are almost always used to read, expand, and manipulate these values. Because of this, date and time manipulation functions are some of the most important functions in the MySQL language.

Table 11.2 lists some commonly used date and time manipulation functions.

Table 11.2 Commonly Used Date and Time Manipulation Functions

Function	Description
`AddDate()`	Adds to a date (days, weeks, and so on)
`AddTime()`	Adds to a time (hours, minutes, and so on)

Function	Description
`CurDate()`	Returns the current date
`CurTime()`	Returns the current time
`Date()`	Returns the date portion of a datetime value
`DateDiff()`	Calculates the difference between two dates
`Date_Add()`	Does date arithmetic
`Date_Format()`	Returns a formatted date or time string
`Day()`	Returns the day portion of a date
`DayOfWeek()`	Returns the day of week for a date
`Hour()`	Returns the hour portion of a time
`Minute()`	Returns the minute portion of a time
`Month()`	Returns the month portion of a date
`Now()`	Returns the current date and time
`Second()`	Returns the second portion of a time
`Time()`	Returns the time portion of a datetime value
`Year()`	Returns the year portion of a date

This would be a good time to revisit data filtering using `WHERE`. Thus far we have filtered data by using `WHERE` clauses that compare numbers and text, but you frequently need to filter data by date. Filtering by date requires some extra care and the use of special MySQL functions.

The first thing to keep in mind is the date format used by MySQL. Whenever you specify a date, whether to insert or update table values or to filter using a `WHERE` clause, the date must be in the format `yyyy-mm-dd`. So, for September 1, 2023, you specify `2023-09-01`. Although other date formats might be recognized, this is the preferred date format because it eliminates ambiguity. (Is 04/05/06 May 4, 2006, or April 5, 2006, or May 6, 2004?)

Tip

Always Use Four-Digit Years MySQL supports two-digit years. For example, it treats the years 00–69 as 2000–2069 and the years 70–99 as 1970–1999. While these might in fact be the intended years, it is far safer to always use a four-digit year so MySQL does not have to make any assumptions for you.

A basic date comparison is simple enough:

Input

```
SELECT cust_id, order_num
FROM orders
WHERE order_date = '2023-09-01';
```

Output

```
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
+---------+-----------+
```

Analysis

The `SELECT` statement worked in this case. It retrieved a single order record—one with an `order_date` of `2023-09-01`.

But is using `WHERE order_date = '2023-09-01'` safe? `order_date` has the datetime datatype. This type stores dates along with time values. The values in our example tables all have times of `00:00:00`, but that might not always be the case. What if order dates are stored using the current date and time (so you not only know the order date but also the time of day the order was placed)? In this case, `WHERE order_date = '2023-09-01'` fails if, for example, the stored `order_date` value is `2023-09-01 11:30:05`. Even though a row with that date is present, it is not retrieved because the `WHERE` match fails.

The solution is to instruct MySQL to only compare the specified date to the date portion of the column instead of using the entire column value. To do this, you must use the `Date()` function. `Date(order_date)` instructs MySQL to extract just the date part of the column, and so a safer `SELECT` statement is as follows:

Input

```
SELECT cust_id, order_num
FROM orders
WHERE Date(order_date) = '2023-09-01';
```

Tip

If You Mean Date, Use `Date()` It's a good practice to use `Date()` if what you want is just the date, even if you know that the column contains only dates. This way, if a date-time value somehow ends up in the table in the future, your SQL won't break. Oh, and yes, there is a `Time()` function, too, and you should use it when you want the time.

Now that you know how to use dates to test for equality, using all of the other operators (introduced in Chapter 6, "Filtering Data") should be self-explanatory.

But one other type of date comparison warrants explanation. What if you wanted to retrieve all orders placed in September 2023? A simple equality test does not work as it matches the day of month, too. There are several solutions, one of which follows:

Input

```
SELECT cust_id, order_num
FROM orders
WHERE Date(order_date) BETWEEN '2023-09-01'
                       AND '2023-09-30';
```

Output

```
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
|   10003 |     20006 |
|   10004 |     20007 |
+---------+-----------+
```

Analysis

Here a `BETWEEN` operator is used to define `2023-09-01` and `2023-09-30` as the range of dates to match.

Here's another solution (one that won't require you to remember how many days are in each month or worry about February in leap years):

Input

```
SELECT cust_id, order_num
FROM orders
WHERE Year(order_date) = 2023
      AND Month(order_date) = 9;
```

Analysis

`Year()` is a function that returns the year from a date (or a datetime). Similarly, `Month()` returns the month from a date. `WHERE Year(order_date) = 2023 AND Month(order_date) = 9` thus retrieves all rows that have an `order_date` in year `2023` and in month `9`.

Tip

How to Ignore Time Using functions like `Year()`, `Month()`, and `Day()` allows you to work with dates without worrying about any time values that may be stored with them.

Numeric Manipulation Functions

Numeric manipulation functions do what it sounds like they do: manipulate numeric data. These functions tend to be used primarily for algebraic, trigonometric, or geometric calculations and, therefore, are not as frequently used as string or date and time manipulation functions.

The ironic thing is that of all the functions found in the major DBMSs, the numeric functions are the ones that are most uniform and consistent. Table 11.3 lists some of the most commonly used numeric manipulation functions.

Table 11.3 Commonly Used Numeric Manipulation Functions

Function	Description
`Abs()`	Returns the absolute value of a specified number
`Cos()`	Returns the trigonometric cosine of a specified angle
`Exp()`	Returns the exponential value of a specified number
`Mod()`	Returns the remainder of a division operation
`Pi()`	Returns the value of pi
`Rand()`	Returns a random number
`Sin()`	Returns the trigonometric sine of a specified angle
`Sqrt()`	Returns the square root of a specified number
`Tan()`	Returns the trigonometric tangent of a specified angle

Summary

In this chapter, you learned how to use SQL's data manipulation functions and paid special attention to working with dates.

Challenges

1. Our store is now online, and customer accounts are being created. Every user needs a login, and the default login will be a combination of a user's name and city. Write a SQL statement that returns customer ID (`cust_id`), customer name (`customer_name`), and `user_login`, which is all uppercase and composed of the first two characters of the customer contact (`cust_contact`) and the first three characters of the customer city (`cust_city`). So, for example, my login (Ben Forta, living in Oak Park) would be `BEOAK`. Here's a hint: For this one, you'll use functions, concatenation, and an alias.
2. Write a SQL statement to return the order number (`order_num`) and order date (`order_date`) for every order placed in October 2023 and sort the list by order date.

12

Summarizing Data

In this chapter, you will learn what the SQL aggregate functions are and how to use them to summarize table data.

Using Aggregate Functions

It is often necessary to summarize data without actually retrieving it all, and MySQL provides special functions for this purpose. You can use these functions in MySQL queries to retrieve data for analysis and reporting purposes. These are some examples of this type of retrieval:

- Determining the number of rows in a table (or the number of rows that meet some condition or contain a specific value)
- Obtaining the sum of a group of rows in a table
- Finding the highest, lowest, and average values in a table column (either for all rows or for specific rows)

In each of these examples, you want a summary of the data in a table, not the actual data itself. Therefore, returning the actual table data would be a waste of time and processing resources (not to mention bandwidth). To repeat: All you really want is the summary information.

To facilitate this type of retrieval, MySQL features a set of aggregate functions, some of which are listed in Table 12.1. These functions enable you to perform all the types of retrieval just enumerated.

New Term

Aggregate Function A function that operates on a set of rows to calculate and return a single value.

Table 12.1 SQL Aggregate Functions

Function	Description
`Avg()`	Returns a column's average value
`Count()`	Returns the number of rows in a column
`Max()`	Returns a column's highest value
`Min()`	Returns a column's lowest value
`Sum()`	Returns the sum of a column's values

The use of each of these functions is explained in the following sections.

> **Note**
> **Standard Deviation** A series of standard deviation aggregate functions are also supported by MySQL, but they are not covered in this book.

The Avg() Function

`Avg()` is used to return the average value of a specific column by counting both the number of rows in the table and the sum of their values. `Avg()` can be used to return the average value of all columns or of specific columns or rows.

This first example uses `Avg()` to return the average price of all the products in the `products` table:

► Input

```
SELECT Avg(prod_price) AS avg_price
FROM products;
```

► Output

```
+-----------+
| avg_price |
+-----------+
| 16.133571 |
+-----------+
```

► Analysis

This `SELECT` statement returns a single value, `avg_price`, that contains the average price of all products in the `products` table. `avg_price` is an alias (refer to Chapter 10, "Creating Calculated Fields").

`Avg()` can also be used to determine the average value of specific columns or rows.

The following example returns the average price of products offered by a specific vendor:

Input

```
SELECT Avg(prod_price) AS avg_price
FROM products
WHERE vend_id = 1003;
```

Output

```
+-----------+
| avg_price |
+-----------+
| 13.212857 |
+-----------+
```

Analysis

This `SELECT` statement differs from the previous one only in that this one contains a `WHERE` clause. The `WHERE` clause filters to show only products with a `vend_id` of `1003`, and, therefore, the value returned in `avg_price` is the average of just that vendor's products.

Caution

Individual Columns Only You can use `Avg()` to determine the average of a specific numeric column, and that column name must be specified as the function parameter. To obtain the average value of multiple columns, you need to use multiple `Avg()` functions.

Note

`NULL` Values The `Avg()` function ignores column rows that contain `NULL` values.

The `Count()` Function

`Count()` does what it sounds like it does: It counts. Using `Count()`, you can determine the number of rows in a table or the number of rows that match a specific criterion.

You can use `Count()` in two ways:

- Use `Count(*)` to count the number of rows in a table, whether columns contain values or `NULL` values.
- Use `Count(column)` to count the number of rows that have values in a specific column, ignoring `NULL` values.

This example returns the total number of customers in the `customers` table:

Input

```
SELECT Count(*) AS num_cust
FROM customers;
```

Output

```
+----------+
| num_cust |
+----------+
|        5 |
+----------+
```

Analysis

In this example, `Count(*)` is used to count all rows, regardless of values. The count is returned in `num_cust`.

The following example counts just the customers with email addresses provided:

Input

```
SELECT Count(cust_email) AS num_cust
FROM customers;
```

Output

```
+----------+
| num_cust |
+----------+
|        3 |
+----------+
```

Analysis

This `SELECT` statement uses `Count(cust_email)` to count only rows with a value in the `cust_email` column. In this example, `cust_email` is `3` (meaning that only three of the five customers have email addresses listed).

Note

`NULL` Values The `Count()` function ignores column rows with `NULL` values in them if a column name is specified but not if the asterisk (`*`) is used.

The `Max()` Function

`Max()` returns the highest value in a specified column. `Max()` requires that the column name be specified, as shown here:

Input

```
SELECT Max(prod_price) AS max_price
FROM products;
```

Output

```
+-----------+
| max_price |
+-----------+
|     55.00 |
+-----------+
```

Analysis

Here `Max()` returns the price of the most expensive item in the `products` table.

Tip

Using `Max()` with Non-numeric Data Although `Max()` is usually used to find the highest numeric or date values, MySQL allows it to be used to return the highest value in any column—including textual columns. When used with textual data, `Max()` returns the row that would be the last row if the data were sorted by that column.

Note

`NULL` Values The `Max()` function ignores column rows with `NULL` values in them.

The `Min()` Function

`Min()` is exactly the opposite of `Max()`: It returns the lowest value in a specified column. Like `Max()`, `Min()` requires that the column name be specified, as shown here:

Input

```
SELECT Min(prod_price) AS min_price
FROM products;
```

Output

```
+-----------+
| min_price |
+-----------+
| 2.50      |
+-----------+
```

Analysis

Here `Min()` returns the price of the least expensive item in the `products` table.

Tip

Using `Min()` with Non-numeric Data As with the `Max()` function, MySQL allows `Min()` to be used to return the lowest value in any columns, including textual columns. When used with textual data, `Min()` returns the row that would be first if the data were sorted by that column.

> **Note**
> **NULL Values** The `Min()` function ignores column rows with `NULL` values in them.

The Sum() Function

`Sum()` is used to return the sum (total) of the values in a specific column. Here is an example to demonstrate this. The `orderitems` table contains the actual items in an order, and each item has an associated `quantity`. The total number of items ordered (the sum of all the `quantity` values) can be retrieved as follows:

Input

```
SELECT Sum(quantity) AS items_ordered
FROM orderitems
WHERE order_num = 20005;
```

Output

```
+---------------+
| items_ordered |
+---------------+
| 19            |
+---------------+
```

Analysis

The function `Sum(quantity)` returns the sum of all the item quantities in an order, and the `WHERE` clause ensures that just the right order items are included.

`Sum()` can also be used with the mathematical operators introduced in Chapter 10. In this next example, the total order amount is retrieved by totaling `item_price*quantity` for each item:

Input

```
SELECT SUM(item_price*quantity) AS total_price
FROM orderitems
WHERE order_num = 20005;
```

Output

```
+-------------+
| total_price |
+-------------+
|      149.87 |
+-------------+
```

Analysis

The function `Sum(item_price*quantity)` returns the sum of all the expanded prices in an order, and again the `WHERE` clause ensures that just the correct order items are included.

Tip

Performing Calculations on Multiple Columns All the aggregate functions can be used to perform calculations on multiple columns using the standard mathematical operators, as shown in this example.

Note

`NULL` Values The `Sum()` function ignores column rows with `NULL` values in them.

Aggregates on Distinct Values

The five aggregate functions can all be used in two ways:

- To perform calculations on all rows, specify the `ALL` argument or specify no argument at all (because `ALL` is the default behavior).
- To include only unique values, specify the `DISTINCT` argument.

Tip

`ALL` Is the Default The `ALL` argument need not be specified because it is the default behavior. If `DISTINCT` is not specified, `ALL` is assumed.

The following example uses the `Avg()` function to return the average product price offered by a specific vendor. It is the same `SELECT` statement used in the previous example, but here the `DISTINCT` argument is used so the average takes into account only unique prices:

Input

```
SELECT Avg(DISTINCT prod_price) AS avg_price
FROM products
WHERE vend_id = 1003;
```

Output

```
+-----------+
| avg_price |
+-----------+
| 15.998000 |
+-----------+
```

Analysis

As you can see, in this example, `avg_price` is higher when `DISTINCT` is used because there are multiple items with the same lower price. Excluding them raises the average price.

Caution

No DISTINCT with Wildcards DISTINCT can be used with Count() only if a column name is specified. DISTINCT cannot be used with Count(*), and so Count(DISTINCT *) is not allowed and generates an error. Similarly, DISTINCT must be used with a column name and not with a calculation or an expression.

Tip

Using DISTINCT with Min() and Max() Although you can technically use DISTINCT with Min() and Max(), there is actually no value in doing so. The minimum and maximum values in a column are the same whether only distinct values are included or not.

Combining Aggregate Functions

All the examples of aggregate functions used thus far have involved a single function. But actually, SELECT statements may contain as few or as many aggregate functions as needed. Look at this example:

Input

```
SELECT Count(*) AS num_items,
       Min(prod_price) AS price_min,
       Max(prod_price) AS price_max,
       Avg(prod_price) AS price_avg
FROM products;
```

Output

```
+-----------+-----------+-----------+-----------+
| num_items | price_min | price_max | price_avg |
+-----------+-----------+-----------+-----------+
|        14 |      2.50 |     55.00 | 16.133571 |
+-----------+-----------+-----------+-----------+
```

Analysis

Here a single SELECT statement performs four aggregate calculations in one step and returns four values (the number of items in the products table and the highest, lowest, and average product prices).

Tip

Naming Aliases When specifying the name of an alias to contain the results of an aggregate function, try to not use the name of an actual column in the table. Although there is nothing actually illegal about doing so, using unique names makes your SQL easier to understand and work with (and troubleshoot in the future).

Summary

Aggregate functions are used to summarize data. MySQL supports a range of aggregate functions, all of which can be used in multiple ways to return just the results you need. These functions are designed to be highly efficient, and they usually return results far more quickly than you could calculate them yourself within your own client application.

Challenges

1. Write a SQL statement to determine the total number of items sold (using the `quantity` column in `OrderItems`).
2. Modify the statement you just created to determine the total number of product item (`prod_item`) `BR01` sold.
3. This is a bit of a silly one, but imagine that a customer wants to buy one of every single item in the `products` table. Write a SQL statement to calculate the total price.
4. Write a SQL statement to determine the price (`prod_price`) of the most expensive item in the `Products` table that costs no more than 10. Name the calculated field `max_price`.

This page intentionally left blank

13

Grouping Data

In this chapter, you'll learn how to group data so you can summarize subsets of table contents. Grouping data involves two `SELECT` statement clauses that you haven't learned about yet: the `GROUP BY` clause and the `HAVING` clause.

Understanding Data Grouping

In the previous chapter, you learned that the SQL aggregate functions can be used to summarize data. By using those functions, you can count rows, calculate sums and averages, and obtain high and low values without having to retrieve all the data.

All the calculations thus far have been performed on all the data in a table or on data that matched a specific `WHERE` clause. As a reminder, the following example returns the number of products offered by vendor 1003:

Input

```
SELECT Count(*) AS num_prods
FROM products
WHERE vend_id = 1003;
```

Output

```
+------------ +
| num_prods   |
+------------ +
|           7 |
+------------ +
```

But what if you want to return the number of products offered by each vendor? Or products offered by vendors who offer a single product? Or products offered by vendors who offer more than 10 products?

This is where groups come into play. Grouping enables you to divide data into logical sets so you can perform aggregate calculations on each group.

Creating Groups

You create groups in MySQL by using the `GROUP BY` clause in a `SELECT` statement. The best way to understand this is to look at an example:

Input

```
SELECT vend_id, Count(*) AS num_prods
FROM products
GROUP BY vend_id;
```

Output

```
+---------+------------ +
| vend_id | num_prods    |
+---------+------------ +
|     1001 |           3 |
|     1002 |           2 |
|     1003 |           7 |
|     1005 |           2 |
+---------+------------ +
```

Analysis

This `SELECT` statement specifies two columns: `vend_id`, which contains the ID of a product's vendor, and `num_prods`, which is a calculated field (created using the `Count(*)` function). The `GROUP BY` clause instructs MySQL to sort the data and group it by `vend_id`. This causes `num_prods` to be calculated once per `vend_id` rather than once for the entire table. As you can see in the output, vendor 1001 has 3 products listed, vendor 1002 has 2 products listed, vendor 1003 has 7 products listed, and vendor 1005 has 2 products listed.

By using `GROUP BY`, you do not have to specify each group to be evaluated and calculated. That happens automatically. The `GROUP BY` clause instructs MySQL to group the data and then perform the aggregate calculation on each group rather than on the entire result set.

Before you use `GROUP BY`, here are some important rules about its use that you need to know:

- A `GROUP BY` clause can contain as many columns as you want. You can nest groups, which means you have more granular control over how data is grouped.
- If you have nested groups in your `GROUP BY` clause, data is summarized at the last specified group. In other words, all the columns specified are evaluated together when grouping is established (so you won't get data back for each individual column level).
- Every column listed in `GROUP BY` must be a retrieved column or a valid expression (but not an aggregate function). If an expression is used in the `SELECT`, that same expression must be specified in `GROUP BY`. You cannot use aliases.
- Aside from the aggregate calculations statements, every column in your `SELECT` statement should be present in the `GROUP BY` clause.

- If the grouping column contains a row with a `NULL` value, `NULL` will be returned as a group. If there are multiple rows with `NULL` values, they'll all be grouped together.
- The `GROUP BY` clause must come after any `WHERE` clause and before any `ORDER BY` clause.

Tip

Using `ROLLUP` To obtain values for each group and at a summary level (for each group), use the `WITH ROLLUP` keyword, as shown here:

```
SELECT vend_id, COUNT(*) AS num_prods
FROM products
GROUP BY vend_id WITH ROLLUP;
```

Filtering Groups

In addition to allowing you to group data by using `GROUP BY`, MySQL also allows you to filter which groups to include and which to exclude. For example, you might want a list of all customers who have made at least two orders. To obtain this data, you must filter based on the complete group, not on individual rows.

You've already seen the `WHERE` clause in action (introduced in Chapter 6, "Filtering Data"). But `WHERE` does not work for grouping because `WHERE` filters specific rows, not groups. As a matter of fact, `WHERE` has no idea what a group is.

So what do you use instead of `WHERE`? MySQL provides yet another clause for this purpose: the `HAVING` clause. `HAVING` is very similar to `WHERE`. In fact, all types of `WHERE` clauses you've learned about thus far can also be used with `HAVING`. The only difference is that `WHERE` filters rows, and `HAVING` filters groups.

Tip

`HAVING` Supports All of `WHERE`'s Operators In Chapter 6 and Chapter 7, "Advanced Data Filtering," you learned about `WHERE` clause conditions (including wildcard conditions and clauses with multiple operators). All the techniques and options you learned about for `WHERE` can also be applied to `HAVING`. The syntax is identical; just the keyword is different.

So how do you filter rows? Look at the following example:

Input

```
SELECT cust_id, Count(*) AS num_orders
FROM orders
GROUP BY cust_id
HAVING Count(*) >= 2;
```

Input

```
+---------+-------------- +
| cust_id | num_orders    |
+---------+-------------- +
|   10001 |           2   |
+---------+-------------- +
```

Analysis

The first three lines of this `SELECT` statement are similar to the statements shown previously. The final line adds a `HAVING` clause that filters on those groups with `Count(*) >= 2` (that is, two or more orders). As you can see, a `WHERE` clause does not work here because the filtering is based on the group aggregate value, not on the values of specific rows.

Note

The Difference Between `HAVING` and `WHERE` Here's another way to look at it: `WHERE` filters before data is grouped, and `HAVING` filters after data is grouped. This is an important distinction; rows that are eliminated by a `WHERE` clause are not included in the group. Eliminating rows could change the calculated values, which in turn could affect which groups are filtered based on the use of those values in the `HAVING` clause.

So is there ever a need to use both `WHERE` and `HAVING` clauses in one statement? Actually, yes, there is. Say that you need a list of all vendors who have 2 or more products priced at 10 or more. To do this, you can add a `WHERE` clause that filters out products that cost less than 10. You then add a `HAVING` clause to filter just the groups with two or more rows in them.

Here's an example:

Input

```
SELECT vend_id, Count(*) AS num_prods
FROM products
WHERE prod_price >= 10
GROUP BY vend_id
HAVING Count(*) >= 2;
```

Output

```
+---------+------------- +
| vend_id | num_prods    |
+---------+------------- +
|    1003 |           4 |
|    1005 |           2 |
+---------+------------- +
```

Analysis

This statement warrants an explanation. The first line is a basic `SELECT` that uses an aggregate function—much like the examples thus far. The `WHERE` clause filters all rows with `prod_price` of at least 10. Data is then grouped by `vend_id`, and then a `HAVING` clause filters just those groups with a count of 2 or more. Without the `WHERE` clause, two extra rows would have been retrieved (vendor 1002, who only sells products all priced under 10, and vendor 1001, who sells 3 products but only one of them is priced greater or equal to 10), as shown here:

Input

```
SELECT vend_id, Count(*) AS num_prods
FROM products
GROUP BY vend_id
HAVING Count(*) >= 2;
```

Output

```
+---------+------------ +
| vend_id | num_prods    |
+---------+------------ +
|     1001 |           3 |
|     1002 |           2 |
|     1003 |           7 |
|     1005 |           2 |
+---------+------------ +
```

Grouping and Sorting

It is important to understand that `GROUP BY` and `ORDER BY` are very different, even though they often accomplish the same thing. Table 13.1 summarizes the differences between them.

Table 13.1 `ORDER BY` Versus `GROUP BY`

`ORDER BY`	`GROUP BY`
Sorts generated output.	Groups rows. The output might not be in group order, however.
Any columns (even columns not selected) may be used.	Only selected columns or expressions columns may be used, and every selected column must be used.
Never required.	Required if using columns (or expressions) with aggregate functions.

The first difference listed in Table 13.1 is extremely important. More often than not, you will find that data grouped using `GROUP BY` will indeed be output in group order. But that is not always the case, and it is not actually required by the SQL specifications. Furthermore, you might actually want it sorted differently than it is grouped.

Just because you group data one way (to obtain group-specific aggregate values) does not mean that you want the output sorted that same way. You should always provide an explicit `ORDER BY` clause as well, even if it is identical to the `GROUP BY` clause.

> **Tip**
>
> **Don't Forget `ORDER BY`** As a rule, any time you use a `GROUP BY` clause, you should also specify an `ORDER BY` clause. That is the only way to ensure that data is sorted properly. Never rely on `GROUP BY` to sort your data.

To demonstrate the use of both `GROUP BY` and `ORDER BY`, let's look at an example. The following `SELECT` statement is similar to the ones shown previously. It retrieves the order number and total order price of all orders with a total price of 50 or more:

Input

```
SELECT order_num,
       Sum(quantity*item_price) AS ordertotal
FROM orderitems
GROUP BY order_num
HAVING Sum(quantity*item_price) >= 50;
```

Output

```
+-----------+-------------- +
| order_num | ordertotal    |
+-----------+-------------- +
|      20005 | 149.87       |
|      20006 | 55.00        |
|      20007 | 1000.00      |
|      20008 | 125.00       |
+-----------+-------------- +
```

To sort the output by order total, all you need to do is add an `ORDER BY` clause, as follows:

Input

```
SELECT order_num,
       Sum(quantity*item_price) AS ordertotal
FROM orderitems
GROUP BY order_num
HAVING Sum(quantity*item_price) >= 50
ORDER BY ordertotal;
```

Output

```
+-----------+-------------- +
| order_num | ordertotal    |
+-----------+-------------- +
|      20006 | 55.00        |
```

```
|      20008 | 125.00       |
|      20005 | 149.87       |
|      20007 | 1000.00      |
+---------- -+------------ +
```

► Analysis

In this example, the `GROUP BY` clause is used to group the data by order number (the `order_num` column) so that the `Sum(*)` function can return the total order price. The `HAVING` clause filters the data so that only orders with a total price of 50 or more are returned. Finally, the output is sorted using the `ORDER BY` clause.

Combining Grouping and Data Summarization

In Chapter 12, "Summarizing Data," you learned how to use functions like `Count()`, `Min()`, `Sum()`, and so on. You can also use these functions when grouping data to perform sophisticated reporting.

For example, the `orders` table contains a list of all orders placed. Each order has an order date in a column (appropriately named) `order_date`. What if you needed to know the best sales month? Determining this requires counting sales per month, which in turn requires extracting the year and month from `order_date`.

Here's the code:

► Input

```
SELECT Year(order_date) AS order_year,
       Month(order_date) AS order_month,
       Count(*) AS orders_placed
FROM orders
GROUP BY order_year, order_month
ORDER BY orders_placed DESC
```

► Output

```
+------------+-------------+---------------+
| order_year | order_month | orders_placed |
+------------+-------------+---------------+
|       2023 |           9 |             3 |
|       2023 |          10 |             2 |
+------------+-------------+---------------+
```

► Analysis

The `Year()` and `Month()` functions extract the year and month, respectively, from `order_date`, and the values are assigned to aliases. `GROUP BY` uses those aliases to group the retrieved data and returns one row per month per year. As data is grouped by year and month, `Count(*)` counts the number of rows for each month, which is exactly what we need. The output is sorted by `ORDER BY orders_placed DESC` so that the months are listed in descending order of sales.

SELECT Clause Ordering

This is probably a good time to review the order in which SELECT statement clauses are to be specified. Table 13.2 lists all the clauses you have learned thus far, in the order they must be used.

Table 13.2 SELECT Clauses and Their Sequence

Clause	Description	Required
SELECT	Columns or expressions to be returned	Yes
FROM	Table to retrieve data from	Only if selecting data from a table
WHERE	Row-level filtering	No
GROUP BY	Group specification	Only if calculating aggregates by group
HAVING	Group-level filtering	No
ORDER BY	Output sort order	No
LIMIT	Number of rows to retrieve	No

Summary

In Chapter 12, you learned how to use the SQL aggregate functions to perform summary calculations on your data. In this chapter, you learned how to use the GROUP BY clause to perform calculations on groups of data and return results for each group. You saw how to use the HAVING clause to filter specific groups. You also learned the difference between ORDER BY and GROUP BY and between WHERE and HAVING.

Challenges

1. The OrderItems table contains the individual items for each order. Write a SQL statement that returns the number of lines (as order_lines) for each order number (order_num) and sort the results by order_lines.
2. Write a SQL statement that returns a field named cheapest_item, which contains the lowest-cost item for each vendor (using prod_price in the Products table) and sort the results from lowest to highest cost.
3. It's important to identify the best customers. Write a SQL statement to return the order number (order_num in the OrderItems table) for every order of at least 100 items.

4. Another way to determine the best customers is based on how much they have spent. Write a SQL statement to return the order number (`order_num` in the `OrderItems` table) for every order with a total price of at least 1000. Here's a hint: For this one, you'll need to calculate and sum the total (`item_price` multiplied by `quantity`). Sort the results by order number.
5. What is wrong with the following SQL statement? (Try to figure it out without running it.)

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY items
HAVING COUNT(*) >= 3
ORDER BY items, order_num;
```

This page intentionally left blank

14

Working with Subqueries

In this chapter, you'll learn what subqueries are and how to use them.

Understanding Subqueries

`SELECT` statements are SQL queries. All the `SELECT` statements you have seen thus far are simple queries: single statements that retrieve data from individual database tables.

New Term

Query Any SQL statement. However, the term is usually used to refer to `SELECT` statements.

SQL also enables you to create *subqueries*: queries that are embedded in other queries. Why would you want to do this? The best way to understand this concept is to look at a couple of examples.

Filtering by Subquery

The database tables used in all the chapters in this book are relational tables. (See Appendix B, "The Example Tables," for a description of each of the tables and their relationships.) Order data is stored in two tables. The `orders` table stores a single row for each order and contains the order number, customer ID, and order date. The individual order items are stored in the related `orderitems` table. The `orders` table does not store any customer information except for the customer ID. The `customers` table stores the other customer information.

Suppose you want a list of all the customers who have ordered item TNT2. What do you have to do to retrieve this information? Here are the steps:

1. Retrieve the order numbers of all orders containing item TNT2.
2. Retrieve the customer ID of every customer who has an order in the list of order numbers returned in step 1.
3. Retrieve the customer information for each customer ID returned in step 2.

Each of these steps can be executed as a separate query. You can use the results returned by one `SELECT` statement to populate the `WHERE` clause of the next `SELECT` statement.

You can also use subqueries to combine all three queries into a single statement.

The `SELECT` statement shown here should be self-explanatory by now. It retrieves the `order_num` column for every order item with `prod_id` of `TNT2`. The output lists the two orders containing this item:

Input

```
SELECT order_num
FROM orderitems
WHERE prod_id = 'TNT2';
```

Output

```
+-----------+
| order_num |
+-----------+
|     20005 |
|     20007 |
+-----------+
```

The next step is to retrieve the customer IDs associated with orders 20005 and 20007. Using the `IN` clause described in Chapter 7, "Advanced Data Filtering," you can create a `SELECT` statement as follows:

Input

```
SELECT cust_id
FROM orders
WHERE order_num IN (20005,20007);
```

Output

```
+---------+
| cust_id |
+---------+
|   10001 |
|   10004 |
+---------+
```

Now, combine the two queries by turning the first (the one that returned the order numbers) into a subquery. Look at the following `SELECT` statement:

Input

```
SELECT cust_id
FROM orders
WHERE order_num IN (SELECT order_num
                    FROM orderitems
                    WHERE prod_id = 'TNT2');
```

Output

```
+---------+
| cust_id |
+---------+
|   10001 |
|   10004 |
+---------+
```

Analysis

Subqueries are always processed starting with the innermost `SELECT` statement and working outward. When this `SELECT` statement is processed, MySQL actually performs two operations. First, it runs the subquery:

```
SELECT order_num
FROM orderitems
WHERE prod_id='TNT2'
```

This query returns the two order numbers `20005` and `20007`. Those two values are then passed to the `WHERE` clause of the outer query in the comma-delimited format required by the `IN` operator. The outer query now becomes the following:

```
SELECT cust_id
FROM orders
WHERE order_num IN (20005,20007)
```

As you can see, the output is correct and exactly the same as the output returned by the previous hard-coded `WHERE` clause.

Tip

Formatting Your SQL `SELECT` statements that contain subqueries can be difficult to read and debug, especially as they grow in complexity. Breaking up the queries over multiple lines and indenting the lines appropriately, as shown here, can greatly simplify working with subqueries.

You now have the IDs of all the customers who ordered item TNT2. The next step is to retrieve the customer information for each of those customer IDs. The SQL statement to retrieve the two columns is as follows:

Input

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (10001,10004);
```

Instead of hard-coding those customer IDs, you can turn this WHERE clause into yet another subquery:

Input

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN (SELECT order_num
                                      FROM orderitems
                                      WHERE prod_id = 'TNT2'));
```

Output

```
+----------------+--------------+
| cust_name      | cust_contact |
+----------------+--------------+
| Coyote Inc.    | Y Lee        |
| Yosemite Place | Y Sam        |
+----------------+--------------+
```

Analysis

To execute this SELECT statement, MySQL had to actually perform three SELECT statements. The innermost subquery returned a list of order numbers that were then used as the WHERE clause for the subquery above it. That subquery returned a list of customer IDs that were used as the WHERE clause for the top-level query. The top-level query actually returned the desired data.

As you can see, using subqueries in a WHERE clause enables you to write extremely powerful and flexible SQL statements. There is no limit on the number of subqueries that can be nested, although in practice you will find that performance tells you when you are nesting too deeply.

Caution

Columns Must Match When using a subquery in a WHERE clause (as shown here), make sure that the SELECT statements have the same number of columns as in the WHERE clause. Usually, a single column will be returned by the subquery and matched against a single column, but multiple columns may be used, if needed.

Although usually used in conjunction with the IN operator, subqueries can also be used to test for equality (using =), non-equality (using <>), and so on.

Caution

Subqueries and Performance The code shown here works, and it achieves the desired result. However, using subqueries is not always the most efficient way to perform this type of data retrieval—though sometimes it is. More on this is in Chapter 15, "Joining Tables," where we will revisit this example.

Using Subqueries As Calculated Fields

Another way to use subqueries is in creating calculated fields. Suppose you want to display the total number of orders placed by every customer in your `customers` table. Orders are stored in the `orders` table along with the appropriate customer IDs.

To perform this operation, follow these steps:

1. Retrieve the list of customers from the `customers` table.
2. For each customer retrieved, count the number of associated orders in the `orders` table.

As you learned in the previous two chapters, you can use `SELECT Count(*)` to count rows in a table, and by providing a `WHERE` clause to filter a specific customer ID, you can count just that customer's orders. For example, the following code counts the number of orders placed by customer 10001:

Input

```
SELECT Count(*) AS orders
FROM orders
WHERE cust_id = 10001;
```

To perform this `Count(*)` calculation for each customer, you can use `Count(*)` as a subquery. Look at the following code:

Input

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
        FROM orders
        WHERE orders.cust_id = customers.cust_id) AS orders
FROM customers
ORDER BY cust_name;
```

Output

```
+----------------+------------+--------+
| cust_name      | cust_state | orders |
+----------------+------------+--------+
| Coyote Inc.    | MI         |      2 |
| E Fudd         | IL         |      1 |
| Mouse House    | OH         |      0 |
| Wascals        | IN         |      1 |
| Yosemite Place | AZ         |      1 |
+----------------+------------+--------+
```

Analysis

This `SELECT` statement returns three columns for every customer in the `customers` table: `cust_name`, `cust_state`, and `orders`. `orders` (which is a calculated field that is set by a subquery provided in parentheses). This subquery is executed once for every

customer retrieved. In this example, the subquery is executed five times because five customers are retrieved.

The `WHERE` clause in the subquery is a little different from the `WHERE` clauses used previously because it uses fully qualified column names (first mentioned in Chapter 4, "Retrieving Data"). The following clause tells SQL to compare the `cust_id` in the `orders` table to the one currently being retrieved from the `customers` table:

```
WHERE orders.cust_id = customers.cust_id
```

This type of subquery is called a *correlated subquery*.

New Term

Correlated Subquery A subquery that refers to the outer query.

This syntax—the table name and the column name separated by a period—must be used whenever there is possible ambiguity about column names. Why? Well, let's look at what happens if fully qualified column names are not used:

► Input

```
SELECT cust_name,
       cust_state,
       (SELECT Count(*)
        FROM orders
        WHERE cust_id = cust_id) AS orders
FROM customers
ORDER BY cust_name;
```

► Output

```
+---------------+------------+--------+
| cust_name     | cust_state | orders |
+---------------+------------+--------+
| Coyote Inc.   | MI         |      5 |
| E Fudd        | IL         |      5 |
| Mouse House   | OH         |      5 |
| Wascals       | IN         |      5 |
| Yosemite Place | AZ        |      5 |
+---------------+------------+--------+
```

► Analysis

Obviously, the returned results are incorrect; to see this, compare them to the previous results. Why did this happen? There are two `cust_id` columns, one in `customers` and one in `orders`, and those two columns need to be compared to correctly match orders with the appropriate customers. If you don't fully qualify the column names, MySQL assumes that you are comparing the `cust_id` in the `orders` table to itself. And this

statement always returns the total number of orders in the `orders` table (because MySQL checks to see that every order's `cust_id` matches itself, which it always does, of course):

```
SELECT Count(*) FROM orders WHERE cust_id = cust_id;
```

Although subqueries are extremely useful in constructing this type of `SELECT` statement, care must be taken to properly qualify ambiguous column names.

Note

Always More Than One Solution As explained earlier in this chapter, although the sample code shown here works, this is often not the most efficient way to perform this type of data retrieval. We will revisit this example in Chapter 15.

Tip

Build Queries with Subqueries Incrementally Testing and debugging queries with subqueries can be tricky, particularly as these statements grow in complexity. The safest way to build (and test) queries with subqueries is to do so incrementally, in much the same way that MySQL processes them. Build and test the innermost query first. Then build and test the outer query with hard-coded data, and only after you have verified that it is working, embed the subquery. Then test it again. Keep repeating these steps for each additional query. It will take just a little longer to construct your queries if you do it this way, but it will save you lots of time later when you need to try to figure out why queries are not working. It will also significantly increase the likelihood that the queries will work the first time.

Summary

In this chapter, you learned what subqueries are and how to use them. The most common uses for subqueries are in `WHERE` clauses, in `IN` operators, and for populating calculated columns. You saw examples of both of these types of operations.

Challenges

1. Using a subquery, return a list of customers who bought items priced 10 or more. You'll want to use the `OrderItems` table to find the matching order numbers (`order_num`) and then the `Orders` table to retrieve the customer ID (`cust_id`) for each matched order.
2. You need to know the dates when product BR01 was ordered. Write a SQL statement that uses a subquery to determine which orders (in `OrderItems`) purchased items with `prod_id` of `BR01` and then returns the customer ID (`cust_id`)

and order date (`order_date`) for each of them from the `Orders` table. Sort the results by order date.

3. Now let's make it a bit more challenging. Update the previous challenge to return the customer email (`cust_email` in the `Customers` table) for any customer who purchased an item with `prod_id` of `BR01`. Here's a hint: This involves the `SELECT` statement, the innermost query returning `order_num` from `OrderItems`, and the middle query returning `cust_id` from `Customers`.
4. You need a list of customer IDs with the total amount each customer has ordered. Write a SQL statement that returns the customer ID (`cust_id` in the `Orders` table) and `total_ordered` and uses a subquery to return the total of orders for each customer. Sort the results by the amount spent from greatest to the least. Here's a hint: You've used `SUM()` to calculate order totals previously.
5. Write a SQL statement that retrieves all product names (`prod_name`) from the `Products` table, along with a calculated column named `quant_sold` that contains the total number of this item sold (retrieved using a subquery and `SUM(quantity)` on the `OrderItems` table).

15

Joining Tables

In this chapter, you'll learn what joins are, why they are used, and how to create `SELECT` statements using them.

Understanding Joins

One of SQL's most powerful features is the capability to join tables on-the-fly within data retrieval queries. Joins are some of the most important operations you can perform using SQL `SELECT`, and a good understanding of joins and join syntax is an extremely important part of learning SQL.

Before you can effectively use joins, you must understand relational tables and the basics of relational database design. What follows is by no means complete coverage of the subject, but it should be enough to get you up and running.

Understanding Relational Tables

The best way to understand relational tables is to look at a real-world example.

Suppose you have a database table containing a product catalog, with each catalog item in its own row. The information you store with each item includes a product description and price, along with vendor information about the company that creates the product.

Now suppose you have multiple catalog items created by the same vendor. Where would you store the vendor information—things such as vendor name, address, and contact information? You wouldn't want to store that data along with the products for several reasons:

- Because the vendor information is the same for each product the vendor produces, repeating the information for each product would be a waste of time and storage space.
- If vendor information changes (for example, if the vendor moves or contact info changes), you would need to update every occurrence of the vendor information.
- When data is repeated (that is, when the vendor information is used with each product), there is a high likelihood that the data will not be entered exactly the same way each time. Inconsistent data is extremely difficult to use in reporting.

The key here is that having multiple occurrences of the same data is never a good thing, and this principle is the basis for relational database design. Relational tables are designed so information is split into multiple tables, one for each data type. The tables are related to each other through common values (and thus the *relational* in relational design).

In our example, you can create two tables: one for vendor information and one for product information. The `vendors` table contains all the vendor information, one table row per vendor, along with a unique identifier for each vendor. This value, called a *primary key*, can be a vendor ID or any other unique value. (Primary keys are first mentioned in Chapter 1, "Understanding SQL.")

The `products` table stores only product information and no vendor-specific information other than the vendor ID (the `vendors` table's primary key). This key, called a *foreign key*, relates the `vendors` table to the `products` table, and using this vendor ID enables you to use the `vendors` table to find details about the appropriate vendor.

New Term

Foreign Key A column in one table that contains the primary key values from another table, thus defining the relationships between tables.

What does this do for you? Well, consider the following:

- Vendor information is never repeated, and so time and space are not wasted.
- If vendor information changes, you can update a single record in the `vendors` table. Data in related tables does not change.
- As no data is repeated, the data used is obviously consistent, making data reporting and manipulation much simpler.

The bottom line is that relational data can be stored efficiently and manipulated easily. Because of this, relational databases scale far better than non-relational databases.

New Term

Scale To be able to handle an increasing load without failing. A well-designed database or application is said to *scale well*.

Why Use Joins?

As just explained, breaking data into multiple tables enables more efficient storage, easier manipulation, and greater scalability. But these benefits come with a price.

If data is stored in multiple tables, how can you retrieve that data with a single `SELECT` statement?

The answer is to use a join. Simply put, a *join* is a mechanism used to associate (or join) tables within a `SELECT` statement (thus the name join). By using special syntax, you can join multiple tables so a single set of output is returned, and the join associates the correct rows in each table on-the-fly.

Note

Maintaining Referential Integrity It is important to understand that a join is not a physical entity; in other words, it does not exist in the actual database tables. MySQL creates joins as needed, and a join persists for the duration of the query execution.

When using relational tables, it is important that only valid data is inserted into relational columns. Going back to our example, if products were stored in the `products` table with an invalid vendor ID (one not present in the `vendors` table), those products would be inaccessible because they would not be related to any vendor.

To prevent this from occurring, you can instruct MySQL to allow only valid values (ones present in the `vendors` table) in the vendor ID column in the `products` table. This is known as maintaining *referential integrity*, and is achieved by specifying the primary and foreign keys as part of the table definitions (as will be explained in Chapter 21, "Creating and Manipulating Tables").

Creating a Join

Creating a join is very simple. You must specify all the tables to be included and how they are related to each other. Look at the following example:

Input

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products
WHERE vendors.vend_id = products.vend_id
ORDER BY vend_name, prod_name;
```

Output

```
+-------------+----------------+------------+
| vend_name   | prod_name      | prod_price |
+-------------+----------------+------------+
| ACME        | Bird seed      | 10.00      |
| ACME        | Carrots        | 2.50       |
| ACME        | Detonator      | 13.00      |
| ACME        | Safe           | 50.00      |
| ACME        | Sling          | 4.49       |
| ACME        | TNT (1 stick)  | 2.50       |
| ACME        | TNT (5 sticks) | 10.00      |
| Anvils R Us | .5 ton anvil   | 5.99       |
| Anvils R Us | 1 ton anvil    | 9.99       |
| Anvils R Us | 2 ton anvil    | 14.99      |
| Jet Set     | JetPack 1000   | 35.00      |
| Jet Set     | JetPack 2000   | 55.00      |
| LT Supplies | Fuses          | 3.42       |
| LT Supplies | Oil can        | 8.99       |
+-------------+----------------+------------+
```

► Analysis

In this code, the SELECT statement starts the same way as all the statements you've looked at thus far: by specifying the columns to be retrieved. The big difference here is that two of the specified columns (prod_name and prod_price) are in one table, whereas the other (vend_name) is in another table.

Now look at the FROM clause. Unlike all the prior SELECT statements, this one has two tables listed in the FROM clause: vendors and products. These are the names of the two tables that are being joined in this SELECT statement. The tables are correctly joined with a WHERE clause that instructs MySQL to match vend_id in the vendors table with vend_id in the products table.

Notice that the columns are specified as vendors.vend_id and products.vend_id. Fully qualified column names are required here because if you just specify vend_id, MySQL will not be able to tell which vend_id column you are referring to (as there are two of them, one in each table).

Caution

Fully Qualifying Column Names You must use the fully qualified column name (table name and column name separated by a period) whenever there is possible ambiguity about the column you are referring to. MySQL returns an error message if you refer to an ambiguous column name without fully qualifying it with a table name.

The Importance of the WHERE Clause

It might seem strange to use a WHERE clause to set the join relationship, but actually, there is a very good reason for this. Remember that when tables are joined in a SELECT statement, the relationship is constructed on-the-fly. Nothing in the database table definitions can instruct MySQL how to join the tables. You have to do that yourself. When you join two tables, what you are actually doing is pairing every row in the first table with every row in the second table. The WHERE clause acts as a filter to include only rows that match the specified filter condition—the join condition, in this case. Without the WHERE clause, every row in the first table is paired with every row in the second table, regardless of whether they logically go together.

To understand this, look at the following SELECT statement and its output:

► Input

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products
ORDER BY vend_name, prod_name;
```

► Output

```
+---------------+---------------+------------+
| vend_name     | prod_name     | prod_price |
+---------------+---------------+------------+
| ACME          | .5 ton anvil  | 5.99       |
| ACME          | 1 ton anvil   | 9.99       |
| ACME          | 2 ton anvil   | 14.99      |
```

```
| ACME           | Bird seed      | 10.00      |
| ACME           | Carrots        | 2.50       |
| ACME           | Detonator      | 13.00      |
| ACME           | Fuses          | 3.42       |
| ACME           | JetPack 1000   | 35.00      |
| ACME           | JetPack 2000   | 55.00      |
| ACME           | Oil can        | 8.99       |
| ACME           | Safe           | 50.00      |
| ACME           | Sling          | 4.49       |
| ACME           | TNT (1 stick)  | 2.50       |
| ACME           | TNT (5 sticks) | 10.00      |
| Anvils R Us    | .5 ton anvil   | 5.99       |
| Anvils R Us    | 1 ton anvil    | 9.99       |
| Anvils R Us    | 2 ton anvil    | 14.99      |
| Anvils R Us    | Bird seed      | 10.00      |
| Anvils R Us    | Carrots        | 2.50       |
| Anvils R Us    | Detonator      | 13.00      |
| Anvils R Us    | Fuses          | 3.42       |
| Anvils R Us    | JetPack 1000   | 35.00      |
| Anvils R Us    | JetPack 2000   | 55.00      |
| Anvils R Us    | Oil can        | 8.99       |
| Anvils R Us    | Safe           | 50.00      |
| Anvils R Us    | Sling          | 4.49       |
| Anvils R Us    | TNT (1 stick)  | 2.50       |
| Anvils R Us    | TNT (5 sticks) | 10.00      |
| Furball Inc.   | .5 ton anvil   | 5.99       |
| Furball Inc.   | 1 ton anvil    | 9.99       |
| Furball Inc.   | 2 ton anvil    | 14.99      |
| Furball Inc.   | Bird seed      | 10.00      |
| Furball Inc.   | Carrots        | 2.50       |
| Furball Inc.   | Detonator      | 13.00      |
| Furball Inc.   | Fuses          | 3.42       |
| Furball Inc.   | JetPack 1000   | 35.00      |
| Furball Inc.   | JetPack 2000   | 55.00      |
| Furball Inc.   | Oil can        | 8.99       |
| Furball Inc.   | Safe           | 50.00      |
| Furball Inc.   | Sling          | 4.49       |
| Furball Inc.   | TNT (1 stick)  | 2.50       |
| Furball Inc.   | TNT (5 sticks) | 10.00      |
| Jet Set        | .5 ton anvil   | 5.99       |
| Jet Set        | 1 ton anvil    | 9.99       |
| Jet Set        | 2 ton anvil    | 14.99      |
| Jet Set        | Bird seed      | 10.00      |
| Jet Set        | Carrots        | 2.50       |
| Jet Set        | Detonator      | 13.00      |
| Jet Set        | Fuses          | 3.42       |
| Jet Set        | JetPack 1000   | 35.00      |
| Jet Set        | JetPack 2000   | 55.00      |
| Jet Set        | Oil can        | 8.99       |
```

```
| Jet Set        | Safe           | 50.00      |
| Jet Set        | Sling          | 4.49       |
| Jet Set        | TNT (1 stick)  | 2.50       |
| Jet Set        | TNT (5 sticks) | 10.00      |
| Jouets Et Ours | .5 ton anvil   | 5.99       |
| Jouets Et Ours | 1 ton anvil    | 9.99       |
| Jouets Et Ours | 2 ton anvil    | 14.99      |
| Jouets Et Ours | Bird seed      | 10.00      |
| Jouets Et Ours | Carrots        | 2.50       |
| Jouets Et Ours | Detonator      | 13.00      |
| Jouets Et Ours | Fuses          | 3.42       |
| Jouets Et Ours | JetPack 1000   | 35.00      |
| Jouets Et Ours | JetPack 2000   | 55.00      |
| Jouets Et Ours | Oil can        | 8.99       |
| Jouets Et Ours | Safe           | 50.00      |
| Jouets Et Ours | Sling          | 4.49       |
| Jouets Et Ours | TNT (1 stick)  | 2.50       |
| Jouets Et Ours | TNT (5 sticks) | 10.00      |
| LT Supplies    | .5 ton anvil   | 5.99       |
| LT Supplies    | 1 ton anvil    | 9.99       |
| LT Supplies    | 2 ton anvil    | 14.99      |
| LT Supplies    | Bird seed      | 10.00      |
| LT Supplies    | Carrots        | 2.50       |
| LT Supplies    | Detonator      | 13.00      |
| LT Supplies    | Fuses          | 3.42       |
| LT Supplies    | JetPack 1000   | 35.00      |
| LT Supplies    | JetPack 2000   | 55.00      |
| LT Supplies    | Oil can        | 8.99       |
| LT Supplies    | Safe           | 50.00      |
| LT Supplies    | Sling          | 4.49       |
| LT Supplies    | TNT (1 stick)  | 2.50       |
| LT Supplies    | TNT (5 sticks) | 10.00      |
+----------------+----------------+------------+
```

► Analysis

This output shows every product matched with every vendor, including products with the incorrect vendor (and even vendors with no products at all). This is a *Cartesian product*, and it is seldom what you want.

New Term

Cartesian Product The results returned by a table relationship without a join condition. The number of rows retrieved is the number of rows in the first table multiplied by the number of rows in the second table.

Caution

Don't Forget the `WHERE` Clause Make sure all your joins have `WHERE` clauses, or MySQL returns far more data than you want. Similarly, make sure your `WHERE` clauses are correct. An incorrect filter condition causes MySQL to return incorrect data.

Note

Cross Joins Sometimes you'll hear the type of join that returns a Cartesian product referred to as a *cross join*.

Inner Joins

The type of join you have been using so far—a join based on the testing of equality between two tables—is called an *equijoin*. This kind of join is also called an *inner join*. In fact, you may use slightly different syntax for these joins, specifying the type of join explicitly. The following `SELECT` statement returns exactly the same data as the preceding example:

► Input

```
SELECT vend_name, prod_name, prod_price
FROM vendors INNER JOIN products
 ON vendors.vend_id = products.vend_id;
```

► Analysis

The `SELECT` in the statement is the same as the preceding `SELECT` statement, but the `FROM` clause is different. Here the relationship between the two tables is part of the `FROM` clause specified as `INNER JOIN`. When using this syntax, the join condition is specified using the special `ON` clause instead of a `WHERE` clause. The actual condition passed to `ON` is the same as would be passed to `WHERE`.

Note

Which Syntax to Use? Per the ANSI SQL specification, the `INNER JOIN` syntax is preferable. Furthermore, although using the `WHERE` clause to define joins is simpler, using explicit join syntax ensures that you will never forget the join condition, and it can affect performance, too (in some cases). But, that said, the simpler `WHERE` clause is supported, so feel free to use it if you prefer.

Joining Multiple Tables

SQL imposes no limit to the number of tables that may be joined in a `SELECT` statement. The basic rules for creating a join remain the same: First list all the tables and then define the relationships between them. Here is an example:

Input

```
SELECT prod_name, vend_name, prod_price, quantity
FROM orderitems, products, vendors
WHERE products.vend_id = vendors.vend_id
  AND orderitems.prod_id = products.prod_id
  AND order_num = 20005;
```

Output

```
+----------------+-------------+------------+----------+
| prod_name      | vend_name   | prod_price | quantity |
+----------------+-------------+------------+----------+
| .5 ton anvil   | Anvils R Us |       5.99 |       10 |
| 1 ton anvil    | Anvils R Us |       9.99 |        3 |
| TNT (5 sticks) | ACME        |      10.00 |        5 |
| Bird seed      | ACME        |      10.00 |        1 |
+----------------+-------------+------------+----------+
```

Analysis

This example displays the items in order number 20005. Order items are stored in the `orderitems` table. Each product is stored by its product ID, which refers to a product in the `products` table. The products are linked to the appropriate vendors in the `vendors` table by vendor ID, which is stored with each product record. The `FROM` clause here lists the three tables, and the `WHERE` clause defines both of those join conditions. An additional `WHERE` condition is then used to filter just the items for order 20005.

Here is the `INNER JOIN` version of the same `SELECT` statement:

Input

```
SELECT prod_name, vend_name, prod_price, quantity
FROM vendors
INNER JOIN products
   ON vendors.vend_id = products.vend_id
INNER JOIN orderitems
   ON orderitems.prod_id = products.prod_id
WHERE order_num = 20005;
```

Analysis

In this version, the `SELECT` is the same, as is the final `WHERE` clause. The difference is in how the tables are defined and joined together. Here, `vendors` is the `FROM` table, and the two additional tables are included and joined using `INNER JOIN` and an `ON` to define the relationship. The syntax is different, but the results will be identical.

Caution

Performance Considerations MySQL processes joins at runtime, relating each table as specified. This process can become very resource intensive, so be careful not to join tables unnecessarily. The more tables you join, the more performance degrades.

Now would be a good time to revisit an example from Chapter 14, "Working with Subqueries." As you will recall, this `SELECT` statement returns a list of customers who ordered product TNT2:

Input

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN (SELECT order_num
                                      FROM orderitems
                                      WHERE prod_id = 'TNT2'));
```

As mentioned in Chapter 14, using subqueries might not always be the most efficient way to perform complex `SELECT` operations, and so, as promised, here is the same query using joins:

Input

```
SELECT cust_name, cust_contact
FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
  AND orderitems.order_num = orders.order_num
  AND prod_id = 'TNT2';
```

Output

```
+---------------+--------------+
| cust_name     | cust_contact |
+---------------+--------------+
| Coyote Inc.   | Y Lee        |
| Yosemite Place | Y Sam       |
+---------------+--------------+
```

Analysis

As explained in Chapter 14, returning the data needed in this query requires the use of three tables. But instead of using them within nested subqueries, here two joins are used to connect the tables. There are three `WHERE` clause conditions here. The first two connect the tables in the join, and the last one filters the data for product TNT2.

For the sake of comparison, here is the `INNER JOIN` version of the same statement:

Input

```
SELECT cust_name, cust_contact
FROM customers
INNER JOIN orders
 ON customers.cust_id = orders.cust_id
INNER JOIN orderitems
 ON orderitems.order_num = orders.order_num
WHERE prod_id = 'TNT2';
```

Output

```
+---------------+--------------+
| cust_name     | cust_contact |
+---------------+--------------+
| Coyote Inc.   | Y Lee        |
| Yosemite Place | Y Sam       |
+---------------+--------------+
```

Tip

It Pays to Experiment As you can see, there is often more than one way to perform any given SQL operation. And there is rarely a definitive right or wrong way. Performance can be affected by the type of operation, the amount of data in the tables, whether indexes and keys are present, and a whole slew of other criteria. Therefore, it is often worth experimenting with different selection mechanisms to find the one that works best in a situation.

Summary

Joins are some of the most important and powerful operations in SQL. To use them effectively, you need a basic understanding of relational database design. In this chapter, you learned some of the basics of relational database design as an introduction to learning about joins. You also learned how to create an equijoin (also known as an inner join), which is the most commonly used form of join. In the next chapter, you'll learn how to create other types of joins.

Challenges

1. Write a SQL statement to return the customer name (`cust_name`) from the `Customers` table and related order numbers (`order_num`) from the `Orders` table and sort the result by customer name and then by order number. Actually, try this one twice: once using simple equijoin syntax and then using `INNER JOIN`.

2. Let's make the previous challenge more useful. In addition to returning the customer name and order number, add a third column named `OrderTotal` that contains the total price of each order. There are two ways to do this: You can create the `OrderTotal` column using a subquery on the `OrderItems` table, or you can join the `OrderItems` table to the existing tables and use an aggregate function. Here's a hint: Watch out for where you need to use fully qualified column names.
3. Let's revisit Challenge 2 from Chapter 14. Write a SQL statement that retrieves the dates when product BR01 was ordered, but this time use a join and simple equijoin syntax. The output should be identical to the output in the Chapter 14 challenge.
4. That was fun; let's try it again. Re-create the SQL you wrote for Challenge 3 from Chapter 14 but this time using ANSI `INNER JOIN` syntax. The code you wrote there employed two nested subqueries. To re-create it, you'll need two `INNER JOIN` statements, each formatted like the `INNER JOIN` example earlier in this chapter. And don't forget the `WHERE` clause to filter by `prod_id`.
5. One more, and to make things more fun, we'll mix joins, aggregates functions, and grouping, too. Ready? Back in Chapter 13, I challenged you to find all order numbers with a value of 1000 or more. Those results are useful, but what would be even more useful is the names of the customers who placed orders of at least that amount. So, write a SQL statement that uses joins to return the customer name (`cust_name`) from the `Customers` table and the total price of every order from the `OrderItems` table. Here's a hint: To join those tables, you'll also need to include the `Orders` table (because `Customers` is not related directly to `OrderItems`, `Customers` is related to `Orders`, and `Orders` is related to `OrderItems`). Don't forget `GROUP BY` and `HAVING` and be sure to sort the results by customer name. You can use simple equijoin or ANSI `INNER JOIN` syntax for this one. Or, if you are feeling brave, try writing it both ways.

This page intentionally left blank

16

Creating Advanced Joins

In this chapter, you'll learn all about additional join types—what they are and how to use them. You'll also learn how to use table aliases and how to use aggregate functions with joined tables.

Using Table Aliases

In Chapter 10, "Creating Calculated Fields," you learned how to use aliases to refer to retrieved table columns. The code to alias a column looks like this:

Input

```
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country), ')')
   AS vend_title
FROM vendors
ORDER BY vend_name;
```

In addition to using aliases for column names and calculated fields, SQL also enables you to alias table names. There are two primary reasons to do this:

- To shorten the SQL syntax
- To enable multiple uses of the same table within a single `SELECT` statement

Take a look at the following `SELECT` statement. It is basically the same as a statement used in the previous chapter, but it has been modified to use aliases:

Input

```
SELECT cust_name, cust_contact
FROM customers AS c, orders AS o, orderitems AS oi
WHERE c.cust_id = o.cust_id
  AND oi.order_num = o.order_num
  AND prod_id = 'TNT2';
```

Analysis

Notice that the three tables in the `FROM` clauses all have aliases. `customers AS c` establishes `c` as an alias for `customers`, and so on. This alias enables you to use the abbreviated `c` instead of the full text `customers`. In this example, the table aliases are used only in the

WHERE clause, but aliases are not limited to just WHERE. You can use aliases in the SELECT list, the ORDER BY clause, and any other part of a statement as well.

For the sake of comparison, here is the INNER JOIN version:

Input

```
SELECT cust_name, cust_contact
FROM customers AS c
INNER JOIN orders AS o
 ON c.cust_id = o.cust_id
INNER JOIN orderitems AS oi
 ON oi.order_num = o.order_num
WHERE prod_id = 'TNT2';
```

Analysis

As before, aliases are defined by using AS for each table. This time, two of them are in INNER JOIN clauses.

Note

Table Aliases Are DBMS Only It is also worth noting that table aliases are only used during query execution. Unlike column aliases, table aliases are never returned to the client.

Using Different Join Types

So far, you have used only simple joins known as inner joins, or *equijoins*. Let's now take a look at three additional join types: self-joins, natural joins, and outer joins.

Self-Joins

As mentioned earlier, one of the primary reasons to use table aliases is to be able to refer to the same table more than once in a single SELECT statement. Let's look at an example that demonstrates this.

Suppose that a problem has been found with a product (item id DTNTR), and you want a list of all the products made by that vendor so you can determine whether the same problem applies to them, too. This query requires that you first find out which vendor creates item DTNTR and then find which other products are made by the same vendor. The following is one way to approach this problem:

Input

```
SELECT prod_id, prod_name
FROM products
WHERE vend_id = (SELECT vend_id
                 FROM products
                 WHERE prod_id = 'DTNTR');
```

Output

```
+---------+----------------+
| prod_id | prod_name      |
+---------+----------------+
| DTNTR   | Detonator      |
| FB      | Bird seed      |
| FC      | Carrots        |
| SAFE    | Safe           |
| SLING   | Sling          |
| TNT1    | TNT (1 stick)  |
| TNT2    | TNT (5 sticks) |
+---------+----------------+
```

Analysis

This first solution uses subqueries. The inner `SELECT` statement does a simple retrieval to return the `vend_id` of the vendor that makes item DTNTR. That ID is the one used in the `WHERE` clause of the outer query so that all items produced by that vendor are retrieved. (You learned all about subqueries in Chapter 14, "Working with Subqueries." Refer to that chapter for more information.)

Now look at the same query using a join:

Input

```
SELECT p1.prod_id, p1.prod_name
FROM products AS p1, products AS p2
WHERE p1.vend_id = p2.vend_id
  AND p2.prod_id = 'DTNTR';
```

Output

```
+---------+----------------+
| prod_id | prod_name      |
+---------+----------------+
| DTNTR   | Detonator      |
| FB      | Bird seed      |
| FC      | Carrots        |
| SAFE    | Safe           |
| SLING   | Sling          |
| TNT1    | TNT (1 stick)  |
| TNT2    | TNT (5 sticks) |
+---------+----------------+
```

Analysis

The two tables needed in this query are actually the same table, and so the `products` table appears in the `FROM` clause twice. Although this is perfectly legal, any references to the `products` table would be ambiguous because MySQL would not know to which instance of the `products` table you were referring.

To resolve this problem, you can use table aliases. The first occurrence of `products` has the alias `p1`, and the second has the alias `p2`. Now those aliases can be used as table

names. The `SELECT` statement, for example, uses the `p1` prefix to explicitly state the full name of the desired columns. If it doesn't, MySQL returns an error because there are two columns named `prod_id` and `prod_name`. It cannot know which one you want (even though, in truth, they are one and the same). The `WHERE` clause first joins the tables (by matching `vend_id` in `p1` to `vend_id` in `p2`), and then it filters the data by `prod_id` in the second table to return only the desired data.

Here is the same statement using `INNER JOIN` syntax:

Input

```
SELECT p1.prod_id, p1.prod_name
FROM products AS p1
INNER JOIN products AS p2
 ON p1.vend_id = p2.vend_id
WHERE p2.prod_id = 'DTNTR';
```

Tip

Self-Joins Instead of Subqueries Self-joins are often used to replace statements using subqueries that retrieve data from the same table as the outer statement. Although the end result is the same, sometimes these joins execute far more quickly than do subqueries. It is usually worth experimenting with both to determine which performs better in a particular situation.

Natural Joins

Whenever tables are joined, at least one column appears in more than one table (the columns being joined). Standard joins (the inner joins you learned about in the previous chapter) return all data—even multiple occurrences of the same column. A *natural join* simply eliminates those multiple occurrences so only one of each column is returned.

How does it do this? The answer is it doesn't; you do it. A natural join is a join in which you select only columns that are unique. This is typically done using a wildcard (`SELECT *`) for one table and explicit subsets of the columns for all other tables. The following is an example:

Input

```
SELECT c.*, o.order_num, o.order_date,
       oi.prod_id, oi.quantity, OI.item_price
FROM customers AS c, orders AS o, orderitems AS oi
WHERE c.cust_id = o.cust_id
  AND oi.order_num = o.order_num
  AND prod_id = 'FB';
```

Analysis

In this example, a wildcard is used for the first table only. All other columns are explicitly listed so that no duplicate columns are retrieved.

> **Tip**
>
> **Inner Joins and Natural Joins** The truth is, every inner join you have created thus far is actually a natural join, and you will probably never even need an inner join that is not a natural join.

Outer Joins

Most joins relate rows in one table with rows in another. But occasionally, you want to include rows that have no related rows. For example, you might use joins to accomplish the following tasks:

- Count how many orders each customer placed, including customers who have yet to place an order
- List all products with order quantities, including products not ordered by anyone
- Calculate average sale sizes, taking into account customers who have not yet placed an order

In each of these examples, the join includes table rows that have no associated rows in the related table. This type of join is called an *outer join*.

The following `SELECT` statement is a simple inner join. It retrieves a list of all customers and their orders:

Input

```
SELECT customers.cust_id, orders.order_num
FROM customers INNER JOIN orders
 ON customers.cust_id = orders.cust_id;
```

Outer join syntax is similar. To retrieve a list of all customers, including those who have placed no orders, you can use the following:

Input

```
SELECT customers.cust_id, orders.order_num
FROM customers LEFT OUTER JOIN orders
 ON customers.cust_id = orders.cust_id;
```

Output

```
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
|   10001 |     20009 |
|   10002 |      NULL |
|   10003 |     20006 |
|   10004 |     20007 |
|   10005 |     20008 |
+---------+-----------+
```

► Analysis

Like the inner join shown in the previous chapter, this SELECT statement uses the keyword OUTER JOIN to specify the join type (instead of the WHERE clause). But unlike inner joins, which relate rows in both tables, outer joins also include rows with no related rows. When using OUTER JOIN syntax, you must use the RIGHT keyword or LEFT keyword to specify the table from which to include all rows; you use RIGHT for the one on the right of OUTER JOIN and LEFT for the one on the left. The previous example uses LEFT OUTER JOIN to select all the rows from the table on the left in the FROM clause (the customers table). To select all the rows from the table on the right, you use RIGHT OUTER JOIN, as shown in this example:

► Input

```
SELECT customers.cust_id, orders.order_num
FROM customers RIGHT OUTER JOIN orders
 ON orders.cust_id = customers.cust_id;
```

Note

No *= For an OUTER JOIN, you need to use ANSI syntax. MySQL does not support the use of the simplified *= and =* syntax popularized by other DBMSs.

Tip

Outer Join Types There are two basic forms of outer joins: the left outer join and the right outer join. The only difference between them is the order of the tables that are being related. A left outer join can be turned into a right outer join simply by reversing the order of the tables in the FROM or WHERE clause. Therefore, the two types of outer join can be used interchangeably, and the decision about which one to use is based purely on convenience.

Using Joins with Aggregate Functions

As you learned in Chapter 12, "Summarizing Data," aggregate functions are used to summarize data. Although all the examples of aggregate functions thus far have only summarized data from a single table, these functions can also be used with joins. To demonstrate this, let's look at an example.

You want to retrieve a list of all customers and the number of orders that each has placed. The following code uses the Count() function to achieve this:

► Input

```
SELECT customers.cust_name,
       customers.cust_id,
       Count(orders.order_num) AS num_ord
FROM customers INNER JOIN orders
 ON customers.cust_id = orders.cust_id
GROUP BY customers.cust_id;
```

Output

```
+----------------+---------+---------+
| cust_name      | cust_id | num_ord |
+----------------+---------+---------+
| Coyote Inc.    |   10001 |       2 |
| Wascals        |   10003 |       1 |
| Yosemite Place |   10004 |       1 |
| E Fudd         |   10005 |       1 |
+----------------+---------+---------+
```

Analysis

This `SELECT` statement uses `INNER JOIN` to relate the `customers` and `orders` tables to each other. The `GROUP BY` clause groups the data by customer, and the function call `COUNT(orders.order_num)` counts the number of orders for each customer and returns it as `num_ord`.

Aggregate functions can be used just as easily with other join types. See the following example:

Input

```
SELECT customers.cust_name,
       customers.cust_id,
       Count(orders.order_num) AS num_ord
FROM customers LEFT OUTER JOIN orders
 ON customers.cust_id = orders.cust_id
GROUP BY customers.cust_id;
```

Output

```
+----------------+---------+---------+
| cust_name      | cust_id | num_ord |
+----------------+---------+---------+
| Coyote Inc.    |   10001 |       2 |
| Mouse House    |   10002 |       0 |
| Wascals        |   10003 |       1 |
| Yosemite Place |   10004 |       1 |
| E Fudd         |   10005 |       1 |
+----------------+---------+---------+
```

Analysis

This example uses a left outer join to include all customers—even those who have not placed any orders. The output shows that customer `Mouse House` (with 0 orders) is also included this time.

Using Joins and Join Conditions

Before wrapping up this two-chapter discussion of joins, it is worthwhile to summarize some key points regarding joins and their use:

- Pay careful attention to the type of join being used. More often than not, you'll want an inner join, but there are often valid uses for outer joins, too.

- Make sure you use the correct join condition, or you'll get incorrect data returned.
- Make sure you always provide a join condition, or you'll end up with the Cartesian product.
- You may include multiple tables in a join and even have different join types for each one. Although this is legal and often useful, make sure you test each join separately before testing them together. Doing so makes troubleshooting far simpler.

Summary

This chapter is a continuation of the previous chapter on joins. In this chapter you learned how and why to use aliases, and you learned about the different join types and various forms of syntax used with each of them. You also learned how to use aggregate functions with joins, and you learned some important do's and don'ts to keep in mind when working with joins.

Challenges

1. Write a SQL statement using `INNER JOIN` to retrieve customer names (`cust_name` in `Customers`) and all order numbers (`order_num` in `Orders`) for each.
2. Modify the SQL statement you just created to list all customers, even those with no orders.
3. Use `OUTER JOIN` to join the `Products` and `OrderItems` tables and return a sorted list of product names (`prod_name`) and the order numbers (`order_num`) associated with each.
4. Modify the SQL statement created in the previous challenge so that it returns the total number of orders for each item (as opposed to the order numbers).
5. Write a SQL statement that lists vendors (`vend_id` in `Vendors`) and the number of products they have available, including vendors with no products. You'll want to use `OUTER JOIN` and the `COUNT()` aggregate function to count the number of products for each of them in the `Products` table. Pay attention: The `vend_id` column appears in multiple tables, so any time you refer to it, you'll need to fully qualify it.

17

Combining Queries

In this chapter, you'll learn how to use the `UNION` operator to combine multiple `SELECT` statements into one result set.

Understanding Combined Queries

A SQL query usually contains a single `SELECT` statement that returns data from one or more tables. MySQL also enables you to perform multiple queries (multiple `SELECT` statements) and return the results as a single query result set. These combined queries are usually known as *unions* or *compound queries*.

There are basically two scenarios in which you'd use combined queries:

- To return similarly structured data from different tables in a single query
- To perform multiple queries against a single table and return the data as one query

Tip

Combining Queries and Multiple `WHERE` Conditions For the most part, combining two queries to the same table accomplishes the same thing as using a single query with multiple `WHERE` clause conditions. In other words, any `SELECT` statement with multiple `WHERE` clauses can also be specified as a combined query, as you'll see in the section that follows. The performance of each of the two techniques, however, can vary based on the queries used. It is always a good idea to experiment to determine which is preferable in a specific situation.

Creating Combined Queries

You combine SQL queries by using the `UNION` operator. By using `UNION`, you can specify multiple `SELECT` statements, and you can combine their results into a single result set.

Using `UNION`

Using `UNION` is simple enough. All you do is specify each `SELECT` statement and place the keyword `UNION` between them.

Let's look at an example. Say that you need a list of all products costing 5 or less. You also want to include all products made by vendors 1001 and 1002, regardless of price. Of course, you can create a `WHERE` clause to do this, but this time you'll use a `UNION` instead.

As just explained, creating a `UNION` involves writing multiple `SELECT` statements. Here are the individual statements:

Input

```
SELECT vend_id, prod_id, prod_price
FROM products
WHERE prod_price <= 5;
```

Output

```
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1003 | FC      |       2.50 |
|    1002 | FU1     |       3.42 |
|    1003 | SLING   |       4.49 |
|    1003 | TNT1    |       2.50 |
+---------+---------+------------+
```

Input

```
SELECT vend_id, prod_id, prod_price
FROM products
WHERE vend_id IN (1001,1002);
```

Output

```
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
+---------+---------+------------+
```

Analysis

The first `SELECT` retrieves all products with a price of no more than 5. The second `SELECT` uses `IN` to find all products made by vendors 1001 and 1002.

You can combine these two statements like this:

Input

```
SELECT vend_id, prod_id, prod_price
FROM products
WHERE prod_price <= 5
```

```
UNION
SELECT vend_id, prod_id, prod_price
FROM products
WHERE vend_id IN (1001,1002);
```

Output

```
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1003 | FC      | 2.50       |
|    1002 | FU1     | 3.42       |
|    1003 | SLING   | 4.49       |
|    1003 | TNT1    | 2.50       |
|    1001 | ANV01   | 5.99       |
|    1001 | ANV02   | 9.99       |
|    1001 | ANV03   | 14.99      |
|    1002 | OL1     | 8.99       |
+---------+---------+------------+
```

Analysis

This statement is made up of both of the previous `SELECT` statements separated by the `UNION` keyword. `UNION` instructs MySQL to execute both `SELECT` statements and combine the output into a single query result set.

As a point of reference, here is the same query using multiple `WHERE` clauses instead of `UNION`:

Input

```
SELECT vend_id, prod_id, prod_price
FROM products
WHERE prod_price <= 5
  OR vend_id IN (1001,1002);
```

In this simple example, `UNION` might actually be more complicated than a `WHERE` clause. But with more complex filtering conditions, or if the data is being retrieved from multiple tables (and not just a single table), `UNION` could make the process much simpler.

UNION Rules

As you can see, unions are very easy to use. But a few rules govern exactly which unions can be combined:

- A union must be composed of two or more `SELECT` statements, each separated by the keyword `UNION`. (So, to combine four `SELECT` statements, three `UNION` keywords would be used.)
- Each query in a union must contain the same columns, expressions, or aggregate functions (although columns need not be listed in the same order).
- Column datatypes must be compatible: They need not be exactly the same type, but they must be of a type that MySQL can implicitly convert (for example, different numeric types or different date types).

Aside from these basic rules and restrictions, unions can be used for any data retrieval tasks.

Including or Eliminating Duplicate Rows

Go back to the preceding section, titled "Using `UNION`," and look at the sample `SELECT` statements used. Notice that when these statements are executed individually, the first `SELECT` statement returns four rows, and the second `SELECT` statement returns five rows. However, when the two `SELECT` statements are combined with `UNION`, only eight rows are returned—not nine.

`UNION` automatically removes any duplicate rows from the query result set; in other words, it behaves just as multiple `WHERE` clause conditions in a single `SELECT` would. Because vendor 1002 creates a product that costs less than 5, that row is returned by both `SELECT` statements. When `UNION` is used, the duplicate row is eliminated.

This is the default behavior of `UNION`, but you can change it if you so desire. If you do, in fact, want all occurrences of all matches returned, you can use `UNION ALL` instead of `UNION`.

Look at the following example:

► Input

```
SELECT vend_id, prod_id, prod_price
FROM products
WHERE prod_price <= 5
UNION ALL
SELECT vend_id, prod_id, prod_price
FROM products
WHERE vend_id IN (1001,1002);
```

► Output

```
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1003 | FC      |       2.50 |
|    1002 | FU1     |       3.42 |
|    1003 | SLING   |       4.49 |
|    1003 | TNT1    |       2.50 |
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
+---------+---------+------------+
```

► Analysis

When you use `UNION ALL`, MySQL does not eliminate duplicates. Therefore, this example returns nine rows, one of them occurring twice.

Tip

UNION Versus WHERE At the beginning of this chapter, I said that UNION almost always accomplishes the same thing as multiple WHERE conditions. UNION ALL is the form of UNION that accomplishes what cannot be done with WHERE clauses. If you do, in fact, want all occurrences of matches for every condition (including duplicates), you must use UNION ALL and not WHERE.

Sorting Combined Query Results

SELECT statement output is sorted using the ORDER BY clause. When you combine queries with UNION, only one ORDER BY clause may be used, and it must occur after the final SELECT statement. There is very little point in sorting part of a result set one way and part another way, and so multiple ORDER BY clauses are not allowed.

The following example sorts the results returned by the previously used union:

► Input

```
SELECT vend_id, prod_id, prod_price
FROM products
WHERE prod_price <= 5
UNION
SELECT vend_id, prod_id, prod_price
FROM products
WHERE vend_id IN (1001,1002)
ORDER BY vend_id, prod_price;
```

► Output

```
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
|    1003 | TNT1    |       2.50 |
|    1003 | FC      |       2.50 |
|    1003 | SLING   |       4.49 |
+---------+---------+------------+
```

► Analysis

This union takes a single ORDER BY clause after the final SELECT statement. Even though the ORDER BY appears to only be a part of that last SELECT statement, MySQL uses it to sort all the results returned by all the SELECT statements.

> **Note**
> **Combining Different Tables** For the sake of simplicity, all of the examples in this chapter combine queries using the same table. However, everything you learned here also applies to using `UNION` to combine queries of different tables.

Summary

In this chapter, you learned how to combine `SELECT` statements with the `UNION` operator. Using `UNION`, you can return the results of multiple queries as one combined query, either including or excluding duplicates. The use of `UNION` can greatly simplify complex `WHERE` clauses and the process of retrieving data from multiple tables.

Challenges

1. Write a SQL statement that combines two `SELECT` statements that retrieve product ID (`prod_id`) and `quantity` from the `OrderItems` table, one filtering for rows with a quantity of exactly 100 and the other filtering for products with an ID that begins with `BNBG`. Sort the results by product ID.
2. Rewrite the SQL statement you just created to instead use a single `SELECT` statement.
3. This one is a little nonsensical, I know, but it does something you learned in this chapter. Write a SQL statement that returns and combines the product name (`prod_name`) from `Products` and the customer name (`cust_name`) from `Customers` and sort the result by product name.
4. What is wrong with the following SQL statement? (Try to figure it out without running it.)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state  = 'MI'
ORDER BY cust_name;
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state = 'IL'
ORDER BY cust_name;
```

18

Full-Text Searching

In this chapter, you'll learn how to use MySQL's full-text searching capabilities to perform sophisticated data querying and selection.

Understanding Full-Text Searching

Note

Not All Engines Support Full-Text Searching As will be explained in Chapter 21, "Creating and Manipulating Tables," MySQL supports the use of several underlying database engines. Not all engines support full-text searching as described in this chapter. The two most commonly used engines are MyISAM and InnoDB; the former supports full-text searching, and the latter does not. This is why although most of the sample tables used in this book were created to use InnoDB, one table (the `productnotes` table) was created to use MyISAM. If you need full-text searching functionality in your applications, keep this in mind.

In Chapter 8, "Using Wildcard Filtering," you were introduced to the `LIKE` keyword that is used to match text (and partial text) using wildcard operators. By using `LIKE`, you can locate rows that contain specific values or parts of values, regardless of the location of those values within the columns.

In Chapter 9, "Searching Using Regular Expressions," we took text-based searching one step further by using regular expressions to match column values. By using regular expressions, it is possible to write very sophisticated matching patterns to locate the desired rows.

But as useful as these search mechanisms are, they have several very important limitations:

- **Performance:** Wildcard and regular expression matching usually requires that MySQL try to match each and every row in a table (and table indexes are rarely of use in these searches). These searches can therefore be very time-consuming as the number of rows to be searched grows.

- **Explicit control:** Using wildcard and regular expression matching, it is very difficult (and not always possible) to explicitly control what is and what is not matched. An example of this is a search specifying a word that must be matched, a word that must not be matched, and a word that may or may not be matched but only if the first word is indeed matched.
- **Intelligent results:** Although wildcard- and regular expression–based searching provide for very flexible searching, neither provides an intelligent way to select results. For example, searching for a specific word would return all rows that contain that word and would not distinguish between rows that contain a single match and those that contain multiple matches (ranking them as potentially better matches). Similarly, searches for a specific word would not find rows that do not contain that word but that do contain other related words.

All of these limitations and more are addressed by full-text searching. When full-text searching is used, MySQL does not need to look at each row individually, analyzing and processing each word individually. Rather, MySQL creates an index of the words (in specified columns), and searches can be made against those words. MySQL can thus quickly and efficiently determine which words match (that is, which rows contain them), which don't, how often they match, and so on.

Using Full-Text Searching

Full-text searching can only be performed on tables with columns specifically indexed to be searchable in this way. Index creation is generally done at table creation time, and the `productnotes` table has a `note_text` column that we've created for just this purpose.

Performing Full-Text Searches

After indexing, full-text searches are performed using two functions: `Match()` to specify the columns to be searched and `Against()` to specify the search expression to be used.

Here is a basic example:

Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('rabbit');
```

Output

```
+----------------------------------------------+
| note_text                                    |
+----------------------------------------------+
| Customer complaint: rabbit has been able to detect
  trap, food apparently less effective now.    |
| Quantity varies, sold by the sack load. All
  guaranteed to be bright and orange, and suitable
  for use as rabbit bait.                      |
+----------------------------------------------+
```

► Analysis

In this example, the `SELECT` statement retrieves a single column, `note_text`. For the `WHERE` clause, a full-text search is performed. `Match(note_text)` instructs MySQL to perform the search against that named column, and `Against('rabbit')` specifies the word `rabbit` as the search text. The two rows that contain the word `rabbit` are returned.

Note

Using the Full `Match()` Specification The value passed to `Match()` must be the same as the one used in the `FULLTEXT()` definition. If multiple columns are specified, all of them must be listed (and in the correct order).

Note

Searches Are Not Case-Sensitive Full-text searches are not case-sensitive unless `BINARY` mode (not covered in this chapter) is used.

The search just performed could just as easily use a `LIKE` clause, as shown here:

► Input

```
SELECT note_text
FROM productnotes
WHERE note_text LIKE '%rabbit%';
```

► Output

```
+--------------------------------------------------------------------------- +
| note_text                                                                  |
+--------------------------------------------------------------------------- |
| Quantity varies, sold by the sack load. All guaranteed to be               |
| bright and orange, and suitable for use as rabbit bait.                    |
| Customer complaint: rabbit has been able to detect trap, food              |
| apparently less effective now.                                             |
+--------------------------------------------------------------------------- +
```

► Analysis

This `SELECT` retrieves the same two rows, but the order is different (although that may not always be the case).

Neither of the two `SELECT` statements contains an `ORDER BY` clause. The latter (using `LIKE`) returns data in no particularly useful order. But the former (using full-text searching) returns data ordered by how well the text matched. Both rows contain the word `rabbit`, but the row that contains the word `rabbit` as the 3rd word ranks higher than the row that contains it as the 20th word. This is important. An important part of full-text searching is the ranking of results. Rows with a higher rank are returned first (as there is a higher degree of likelihood that those are the rows you really want).

To see how ranking works, look at this example:

Input

```
SELECT note_text,
       Match(note_text) Against('rabbit') AS match_rank
FROM productnotes;
```

Output

```
+-------------------------------------------------+------------------------ +
| note_text                                       | match_rank              |
+-------------------------------------------------+------------------------ +
| Customer complaint: Sticks not individually     |       0                 |
| wrapped, too easy to mistakenly detonate all    |                         |
| at once. Recommend individual wrapping.         |                         |
| Can shipped full, refills not available. Need   |       0                 |
| to order new can if refill needed.              |                         |
| Safe is combination locked, combination not     |       0                 |
| provided with safe. This is rarely a problem    |                         |
| as safes are typically blown up or dropped by   |                         |
| customers.                                      |                         |
| Quantity varies, sold by the sack load. All     | 1.5905543170914         |
| guaranteed to be bright and orange, and         |                         |
| suitable for as rabbit bait.                    |                         |
| Included fuses are short and have been known to |       0                 |
| detonate too quickly for some customers. Longer |                         |
| fuses are available (item FU1) and should be    |                         |
| recommended.                                    |                         |
| Matches not included, recommend purchase of     |       0                 |
| matches or detonator (item DTNTR).              |                         |
| Please note that no returns will be accepted if |       0                 |
| safe opened using explosives.                   |                         |
| Multiple customer returns, anvils failing to    |       0                 |
| drop fast enough or falling backwards on        |                         |
| purchaser. Recommend that customer considers    |                         |
| using heavier anvils.                           |                         |
| Item is extremely heavy. Designed for dropping, |      0                  |
| not recommended for use with slings, ropes,     |                         |
| pulleys, or tightropes.                         |                         |
| Customer complaint: rabbit has been able to     | 1.6408053837485         |
| detect trap, food apparently less effective     |                         |
| now.                                            |                         |
| Shipped unassembled, requires common tools      |      0                  |
| (including oversized hammer).                   |                         |
| Customer complaint: Circular hole in safe floor |      0                  |
| can apparently be easily cut with handsaw.      |                         |
| Customer complaint: Not heavy enough to         |      0                  |
| generate flying stars around head of victim.    |                         |
| If being purchased for dropping, recommend      |                         |
| ANV02 or ANV03 instead.                         |                         |
```

```
| Call from individual trapped in safe plummeting       |      0            |
| to the ground, suggests an escape hatch be            |                   |
| added. Comment forwarded to vendor.                   |                   |
+-------------------------------------------------+----------------------- +
```

► Analysis

Here `Match()` and `Against()` are used in the `SELECT` instead of the `WHERE` clause. This causes all rows to be returned (as there is no `WHERE` clause). `Match()` and `Against()` are used to create a calculated column (with the alias `match_rank`) that contains the ranking value calculated by the full-text search. MySQL calculates the ranking based on the number of words in the row, the number of unique words, the total number of words in the entire index, and the number of rows that contain the word.

As you can see, the rows that do not contain the word `rabbit` have a rank of `0` (and were therefore not selected by the `WHERE` clause in the previous example). The two rows that do contain the word `rabbit` each have a rank value, and the one with the word earlier in the text has a higher rank value than the one in which the word appears later. This helps demonstrate how full-text searching eliminates rows (those with a rank of `0`), and how it sorts results (by rank, in descending order).

Note

Ranking Multiple Search Terms If multiple search terms are specified, those that contain the most matching words are ranked higher than those with fewer (or with just a single match).

As you can see, full-text searching offers functionality not available with simple `LIKE` searches. And as data is indexed, full-text searches are considerably faster, too.

Using Query Expansion

Query expansion is used to try to widen the range of returned full-text search results. Consider the following scenario. You want to find all notes with references to `anvils` in them. Only one note contains the word `anvils`, but you also want any other rows that may be related to your search, even if the specific word `anvils` is not contained within them.

This is a job for query expansion. When query expansion is used, MySQL makes two passes through the data and indexes to perform the search:

1. First, MySQL performs a basic full-text search to find all rows that match the search criteria.
2. MySQL examines those matched rows and selects all useful words. (We'll talk about how MySQL figures out what is useful and what is not shortly.)
3. MySQL performs the full-text search again, this time using not just the original criteria but also all of the useful words.

So, by using query expansion, you can find results that might be relevant, even if they don't contain the exact words for which you are looking.

Let's look at an example. First, a simple full-text search, without query expansion:

Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('anvils');
```

Output

```
+--------------------------------------------------------------------------- +
| note_text                                                                  |
+--------------------------------------------------------------------------- +
| Multiple customer returns, anvils failing to drop fast enough or           |
| falling backwards on purchaser. Recommend that customer considers          |
| using heavier anvils.                                                      |
+--------------------------------------------------------------------------- +
```

Analysis

Only one row contains the word `anvils`, so only one row is returned. Here is the same search, this time using query expansion:

Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('anvils' WITH QUERY EXPANSION);
```

Output

```
+--------------------------------------------------------------------------- +
| note_text                                                                  |
+--------------------------------------------------------------------------- +
| Multiple customer returns, anvils failing to drop fast enough or           |
| falling backwards on purchaser. Recommend that customer considers          |
| using heavier anvils.                                                      |
| Customer complaint: Sticks not individually wrapped, too easy to           |
| mistakenly detonate all at once. Recommend individual wrapping.            |
| Customer complaint: Not heavy enough to generate flying stars              |
| around head of victim. If being purchased for dropping, recommend          |
| ANV02 or ANV03 instead.                                                    |
| Please note that no returns will be accepted if safe opened using          |
| explosives.                                                                |
| Customer complaint: rabbit has been able to detect trap, food              |
| apparently less effective now.                                             |
| Customer complaint: Circular hole in safe floor can apparently be          |
| easily cut with handsaw.                                                   |
| Matches not included, recommend purchase of matches or detonator           |
| (item DTNTR).                                                              |
+--------------------------------------------------------------------------- +
```

Analysis

This time, seven rows are returned. The first row contains the word `anvils` and is thus ranked highest. The second row has nothing to do with `anvils`, but because it contains two words that are also in the first row (`customer` and `recommend`), it is retrieved, too. The third row also contains those same two words, but they are further into the text and further apart, and so this row is included but ranked third. And this third row does indeed refer to `anvils` (by their product name).

As you can see, query expansion greatly increases the number of rows returned, but in doing so, it also increases the number of items returned that you might not actually want.

Tip

The More Rows, the Better The more rows in your table (and the more text within those rows), the better the results returned when using query expansion.

Boolean Text Searches

MySQL supports an additional form of full-text searching called *Boolean mode*. In Boolean mode, you can provide specifics about a number of things, including the following:

- Words to be matched
- Words to be excluded (so that if a row contains this word, it will not be returned, even though other specified words are matched)
- Ranking hints (specifying which words are more important than others so they can be ranked higher)
- Expression grouping

Tip

Usable Even Without a `FULLTEXT` Index Boolean mode differs from the full-text search syntax used thus far in that it may be used even if no `FULLTEXT` index is defined. However, this would be a very slow operation, and the performance would degrade further as data volume increased.

To demonstrate what Boolean mode does, here is a simple example:

Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('heavy' IN BOOLEAN MODE);
```

Output

```
+------------------------------------------------------------------------ +
| note_text                                                               |
+------------------------------------------------------------------------ +
| Item is extremely heavy. Designed for dropping, not recommended         |
| for use with slings, ropes, pulleys, or tightropes.                     |
| Customer complaint: Not heavy enough to generate flying stars           |
| around head of victim. If being purchased for dropping, recommend       |
| ANV02 or ANV03 instead.                                                 |
+------------------------------------------------------------------------ +
```

Analysis

This full-text search retrieves all rows containing the word `heavy` (there are two of them). The keywords `IN BOOLEAN MODE` are specified, but no Boolean operators are actually specified, and so the results are the same as if Boolean mode were not specified.

Note

`IN BOOLEAN MODE` Behaves Differently Although the results in this example are the same as they would be without `IN BOOLEAN MODE`, there is an important difference in behavior (even if it does not manifest itself in this particular example). I'll point out the difference later in this chapter.

To match the rows that contain `heavy` but not any word beginning with `rope`, you can use the following:

Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('heavy -rope*' IN BOOLEAN MODE);
```

Output

```
+------------------------------------------------------------------------ +
| note_text                                                               |
+------------------------------------------------------------------------ +
| Customer complaint: Not heavy enough to generate flying stars           |
| around head of victim. If being purchased for dropping, recommend       |
| ANV02 or ANV03 instead.                                                 |
+------------------------------------------------------------------------ +
```

Analysis

This time, only one row is returned. Again, the word `heavy` is matched, but this time `-rope*` instructs MySQL to explicitly exclude any row that contains `rope*` (that is, any word that begins with `rope`, including `ropes`, which is why one of the rows was excluded).

You have now seen two full-text search Boolean operators: - excludes a word and * is the truncation operator (which you can think of as a wildcard used at the end of a word). Table 18.1 lists all of the supported Boolean operators.

TABLE 18.1 Full-Text Boolean Operators

Operator	Description
+	Include the specified word.
-	Exclude the specified word.
>	Include the specified word and increase its ranking value.
<	Include the specified word and decrease its ranking value.
()	Group the specified words into subexpressions (allowing them to be included, excluded, ranked, and so forth as a group).
~	Negate the specified word's ranking value.
*	Use as a wildcard at the end of a word.
""	Use a text phrase (as opposed to a list of individual words) to match for inclusion or exclusion.

Here are some more examples to demonstrate the use of some of these operators:

► Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('+rabbit +bait' IN BOOLEAN MODE);
```

► Analysis

This search matches rows that contain both the words `rabbit` and `bait`.

► Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('rabbit bait' IN BOOLEAN MODE);
```

► Analysis

Without operators specified, this search matches rows that contain at least one instance of `rabbit` or `bait`.

► Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('"rabbit bait"' IN BOOLEAN MODE);
```

► Analysis

This search matches the phrase `rabbit bait` instead of the two words `rabbit` and `bait`.

► Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('>rabbit <carrot' IN BOOLEAN MODE);
```

► Analysis

This search matches both `rabbit` and `carrot`, increasing the rank of the former and decreasing the rank of the latter.

► Input

```
SELECT note_text
FROM productnotes
WHERE Match(note_text)
      Against('+safe +(<combination)' IN BOOLEAN MODE);
```

► Analysis

This search matches the words `safe` and `combination`, lowering the ranking of the latter.

Note

Ranked but Not Sorted In Boolean mode, rows are returned ranked in descending order by score but not sorted.

Full-Text Searching Notes

Before finishing this chapter, here are some important notes pertaining to the use of full-text searching:

- When indexing full-text data, short words are ignored and are excluded from the index. Short words are by default defined as those having three or fewer characters (though this number can be changed, if needed).
- MySQL comes with a built-in list of *stopwords*, which are words that are always ignored when indexing full-text data. This list can be overridden, if needed. (Refer to the MySQL documentation to learn how to accomplish this.)
- Many words appear so frequently that searching on them would be useless (that is, too many results would be returned). To deal with this, MySQL follows a 50% rule: If a word appears in 50% or more rows, it is treated as a stopword and is effectively ignored. (The 50% rule is not used for Boolean mode.)
- Full-text searching never returns any results if there are fewer than three rows in a table (because every word is always in at least 50% of the rows).

- Single quote characters in words are ignored. For example, `don't` is indexed as `dont`.
- Languages that don't have word delimiters (including Japanese and Chinese) do not return full-text results properly.
- As already noted, full-text searching is only supported in the MyISAM database engine.

Note

No Proximity Operators One feature supported by many full-text search engines is proximity searching, which involves searching for words that are near each other (in the same sentence, in the same paragraph, no more than a specific number of words apart, and so on). Proximity operators are not yet supported for MySQL full-text searching, although this is planned for a future release.

Summary

In this chapter, you learned why full-text searching is used and how to use the MySQL `Match()` and `Against()` functions to perform these searches. You also learned about query expansion as a way to increase the chances of finding related matches and how to use Boolean mode for more granular lookup control.

Challenges

1. Write a SQL statement that uses full-text searching to return all rows that contain the word `safe` but that do not contain the word `handsaw`.
2. Write a SQL statement that uses full-text searching to return all rows that contain the words `drop`, `dropped`, `dropping`, and any other word that begins with `drop`.

This page intentionally left blank

19

Inserting Data

In this chapter, you will learn how to insert data into tables by using the SQL `INSERT` statement.

Understanding Data Insertion

`SELECT` is undoubtedly the most frequently used SQL statement (which is why the past 15 chapters are dedicated to it). But there are three other frequently used SQL statements that you should learn. The first one is `INSERT`. (You'll get to the other two in the next chapter.)

As its name suggests, `INSERT` is used to insert (add) rows to a database table. `INSERT` can be used in several ways:

- To insert a single complete row
- To insert a single partial row
- To insert multiple rows
- To insert the results of a query

We'll now look at each of these.

> **Note**
> **`INSERT` and System Security** Use of the `INSERT` statement can be disabled per table or per user by using MySQL security, as explained in Chapter 28, "Managing Security."

Inserting Complete Rows

The simplest way to insert data into a table is to use the basic `INSERT` syntax, which requires that you specify the table name and the values to be inserted into the new row. Here is an example of this:

Input

```
INSERT INTO Customers
VALUES(NULL,
   'Pep E. LaPew',
```

```
    '100 Main Street',
    'Los Angeles',
    'CA',
    '90046',
    'USA',
    NULL,
    NULL);
```

Note

No Output `INSERT` statements usually generate no output.

► Analysis

This example inserts a new customer into the `customers` table. The data to be stored in each table column is specified in the `VALUES` clause, and a value must be provided for every column. If a column has no value (as is the case for the `cust_contact` and `cust_email` columns), the `NULL` value should be used (assuming that the table allows no value to be specified for that column). The columns must be populated in the order in which they appear in the table definition. The first column, `cust_id`, is also `NULL`. This is because MySQL automatically increments that column each time a row is inserted. You wouldn't want to specify a value (that is MySQL's job), and you can't omit the column (because, as mentioned earlier, every column must be listed). Therefore, a `NULL` value is specified; MySQL ignores it and inserts the next available `cust_id` value in its place.

Although this syntax is indeed simple, it is not at all safe and should generally be avoided at all costs. This SQL statement is highly dependent on the order in which the columns are defined in the table. It also depends on information about that order being readily available. Even if the information is available, there is no guarantee that the columns will be in exactly the same order the next time the table is reconstructed. Therefore, writing SQL statements that depend on specific column ordering is very unsafe. If you try to do it, something will inevitably break at some point.

The safer (and unfortunately more cumbersome) way to write the `INSERT` statement is as follows:

► Input

```
INSERT INTO customers(cust_name,
   cust_address,
   cust_city,
   cust_state,
   cust_zip,
   cust_country,
   cust_contact,
   cust_email)
VALUES('Pep E. LaPew',
   '100 Main Street',
   'Los Angeles',
   'CA',
```

```
   '90046',
   'USA',
   NULL,
   NULL);
```

Analysis

This example does exactly the same thing as the previous `INSERT` statement, but this time, the column names are explicitly stated in parentheses after the table name. When the row is inserted, MySQL matches each item in the columns list with the appropriate value in the `VALUES` list. The first entry in `VALUES` corresponds to the first specified column name. The second value corresponds to the second column name, and so on.

Because column names are provided, the items in the `VALUES` list must match the specified column names in the order in which they are specified—and not necessarily in the order that the columns appear in the table. The advantage of this is that, even if the table layout changes, the `INSERT` statement will still work correctly.

Notice that the `NULL` for `cust_id` is not needed, the `cust_id` column is not listed in the column list, and so no value is needed.

The following `INSERT` statement populates all the row columns (just as before), but it does so in a different order. Because the column names are specified, the insertion will work correctly:

Input

```
INSERT INTO customers(cust_name,
   cust_contact,
   cust_email,
   cust_address,
   cust_city,
   cust_state,
   cust_zip,
   cust_country)
VALUES('Pep E. LaPew',
   NULL,
   NULL,
   '100 Main Street',
   'Los Angeles',
   'CA',
   '90046',
   'USA');
```

Tip

Always Use a Columns List Avoid using `INSERT` without explicitly specifying the columns list. Using a columns list will greatly increase the probability that your SQL will continue to function in the event that table changes occur.

Caution

Using VALUES Carefully Regardless of the INSERT syntax being used, the correct number of items must be specified in the VALUES list. If no column names are provided, a value must be present for every table column. If columns names are provided, a value must be present for each listed column. If none is present, an error message will be generated, and the row will not be inserted.

Using this syntax, you can also omit columns. This means you only provide values for some columns and not for others. (You've actually already seen an example of this; cust_id was omitted when column names were explicitly listed.)

Caution

Omitting Columns You may omit columns from an INSERT operation if the table definition allows it. One of the following conditions must exist:

- The column is defined as allowing NULL values (no value at all).
- A default value is specified in the table definition. This means the default value will be used if no value is specified.

If you omit a value from a table that does not allow NULL values and does not have a default, MySQL generates an error message, and the row is not inserted.

Tip

Improve Overall Performance Databases are frequently accessed by multiple clients, and it is MySQL's job to manage which requests are processed and in which order. INSERT operations can be time-consuming (especially if there are many indexes to be updated), and this can hurt the performance of SELECT statements that are waiting to be processed.

If data retrieval is of utmost importance (as it usually is), you can instruct MySQL to lower the priority of your INSERT statement by adding the keyword LOW_PRIORITY in between INSERT and INTO, like this:

```
INSERT LOW_PRIORITY INTO
```

Incidentally, this also applies to the UPDATE and DELETE statements that you'll learn about in the next chapter.

Inserting Multiple Rows

`INSERT` inserts a single row into a table. But what if you need to insert multiple rows? You can simply use multiple `INSERT` statements, and you can even submit them all at once, with each one terminated by a semicolon, like this:

Input

```
INSERT INTO customers(cust_name,
   cust_address,
   cust_city,
   cust_state,
   cust_zip,
   cust_country)
VALUES('Pep E. LaPew',
   '100 Main Street',
   'Los Angeles',
   'CA',
   '90046',
   'USA');
INSERT INTO customers(cust_name,
   cust_address,
   cust_city,
   cust_state,
   cust_zip,
   cust_country)
VALUES('M. Martian',
   '42 Galaxy Way',
   'New York',
   'NY',
   '11213',
   'USA');
```

Or, if the column names and order are identical in the `INSERT` statements, you can combine the statements as follows:

Input

```
INSERT INTO customers(cust_name,
   cust_address,
   cust_city,
   cust_state,
   cust_zip,
   cust_country)
VALUES(
        'Pep E. LaPew',
        '100 Main Street',
        'Los Angeles',
        'CA',
        '90046',
        'USA'
```

```
    ),
    (
      'M. Martian',
      '42 Galaxy Way',
      'New York',
      'NY',
      '11213',
      'USA'
);
```

Analysis

Here a single `INSERT` statement has multiple sets of values, each enclosed within parentheses and separated by commas.

Tip

Improve `INSERT` Performance The technique just described can improve the performance of your database processing, as MySQL processes multiple insertions in a single `INSERT` faster than it processes multiple `INSERT` statements.

Inserting Retrieved Data

`INSERT` is usually used to add a row to a table by using specified values. There is another form of `INSERT` that can be used to insert the result of a `SELECT` statement into a table. This is known as `INSERT SELECT`, and, as its name suggests, it is made up of an `INSERT` statement and a `SELECT` statement.

Suppose you want to merge a list of customers from another table into your `customers` table. Instead of reading one row at a time and inserting it with `INSERT`, you can do the following:

Note

Instructions Needed for the Next Example The following example imports data from a table named `custnew` into the `customers` table. To try this example, you need to create and populate the `custnew` table first. The format of the `custnew` table should be the same as the format of the `customers` table described in Appendix B, "The Example Tables." When populating `custnew`, be sure not to use `cust_id` values that were already used in `customers` (as the subsequent `INSERT` operation will fail if primary key values are duplicated); alternatively, you can just omit that column and have MySQL generate new values during the import process.

Input

```
INSERT INTO customers(cust_id,
    cust_contact,
    cust_email,
    cust_name,
    cust_address,
    cust_city,
    cust_state,
    cust_zip,
    cust_country)
SELECT cust_id,
    cust_contact,
    cust_email,
    cust_name,
    cust_address,
    cust_city,
    cust_state,
    cust_zip,
    cust_country
FROM custnew;
```

Analysis

This example uses `INSERT SELECT` to import all the data from `custnew` into `customers`. Instead of listing the values to be inserted, the `SELECT` statement retrieves them from `custnew`. Each column in the `SELECT` corresponds to a column in the specified columns list. How many rows will this statement insert? It depends on how many rows are in the `custnew` table. If the table is empty, no rows will be inserted (and no error will be generated because the operation is still valid). If the table does, in fact, contain data, all that data is inserted into `customers`.

This example imports `cust_id` (and assumes that you have ensured that `cust_id` values are not duplicated). You could also simply omit that column (from both the `INSERT` and the `SELECT`), and MySQL generates new values.

Tip

Column Names in `INSERT SELECT` This example uses the same column names in both the `INSERT` and `SELECT` statements for simplicity's sake. But there is no requirement that says the column names must match. In fact, MySQL does not even pay attention to the column names returned by `SELECT`. Rather, the column position is used, so the first column in the `SELECT` statement (regardless of its name) is used to populate the first specified table column, and so on. This is very useful when you're importing data from tables that use different column names.

The `SELECT` statement used in an `INSERT SELECT` can include a `WHERE` clause to filter the data to be inserted.

Note

More Examples Looking for more examples of the use of `INSERT`? See the example table population scripts (described in Appendix B) used to create the example tables for this book.

Summary

In this chapter, you learned how to use `INSERT` to insert rows into a database table. You also learned several other ways to use `INSERT` and why explicit column specification is preferred. You also learned how to use `INSERT SELECT` to import rows from another table. In the next chapter, you'll learn how to use `UPDATE` and `DELETE` to further manipulate table data.

Challenges

1. Using `INSERT` and specifying columns, add yourself to the `Customers` table. Explicitly list the columns you are adding and use only the ones you need.
2. Using `INSERT SELECT`, make backup copies of your `Orders` and `OrderItems` tables.

20

Updating and Deleting Data

In this chapter, you will learn how to use the `UPDATE` and `DELETE` statements to further manipulate table data.

Updating Data

To update (modify) data in a table, you use the `UPDATE` statement. `UPDATE` can be used in two ways:

- To update specific rows in a table
- To update all rows in a table

Let's take a look at each of these uses.

Caution

Don't Omit the `WHERE` Clause Take special care when using `UPDATE` because it is very easy to mistakenly update every row in a table. Please read this entire section on `UPDATE` before using this statement.

Note

`UPDATE` and Security Use of the `UPDATE` statement can be restricted and controlled. See Chapter 28, "Managing Security."

The `UPDATE` statement is very easy to use—maybe even too easy. An `UPDATE` statement is made up of three parts:

- The table to be updated
- The column names and their new values
- The filter condition that determines which rows should be updated

Let's take a look at a simple example. Customer 10005 now has an email address, and his record needs to be updated. The following statement performs this update:

Input

```
UPDATE customers
SET cust_email = 'elmer@fudd.com'
WHERE cust_id = 10005;
```

The `UPDATE` statement always begins with the name of the table being updated. In this example, it is the `customers` table. The `SET` command is then used to assign the new value to a column. In this example, the `SET` clause sets the `cust_email` column to the specified value:

```
SET cust_email = 'elmer@fudd.com'
```

The `UPDATE` statement finishes with a `WHERE` clause that tells MySQL which row to update. Without a `WHERE` clause, MySQL would update all the rows in the `customers` table with this new email address—definitely not the desired effect.

Updating multiple columns requires slightly different syntax:

Input

```
UPDATE customers
SET cust_name = 'The Fudds',
    cust_email = 'elmer@fudd.com'
WHERE cust_id = 10005;
```

When updating multiple columns, only a single `SET` command is used, and each *`column = value`* pair is separated by a comma. (No comma is specified after the last column name.) In this example, columns `cust_name` and `cust_email` are both updated for customer 10005.

Tip

Use Subqueries in an `UPDATE` Statement You can use subqueries in `UPDATE` statements, which enables you to update columns with data retrieved using a `SELECT` statement. Refer to Chapter 14, "Working with Subqueries," for more information on subqueries and their uses.

Tip

The `IGNORE` Keyword If your `UPDATE` statement updates multiple rows and an error occurs while updating one or more of those rows, the entire `UPDATE` operation is cancelled (and any rows updated before the error occurred are restored to their original values). To continue processing updates, even if an error occurs, use the `IGNORE` keyword, like this:

```
UPDATE IGNORE customers ...
```

To delete a column's value, you can set it to NULL (assuming that the table is defined to allow NULL values). You can do this as follows:

Input

```
UPDATE customers
SET cust_email = NULL
WHERE cust_id = 10005;
```

Here the NULL keyword is used to save no value to the cust_email column.

Deleting Data

To delete (remove) data from a table, you use the DELETE statement. DELETE can be used in two ways:

- To delete specific rows from a table
- To delete all rows from a table

You'll now take a look at each of these.

Caution

Don't Omit the WHERE Clause Take special care when using DELETE because it is very easy to mistakenly delete every row from a table. Please read this entire section on DELETE before using this statement.

Tip

DELETE and Security Use of the DELETE statement can be restricted and controlled. See Chapter 28.

I stated earlier that UPDATE is very easy to use. The good (and bad) news is that DELETE is even easier to use.

The following statement deletes a single row from the customers table:

Input

```
DELETE FROM customers
WHERE cust_id = 10006;
```

This statement should be self-explanatory. DELETE FROM requires that you specify the name of the table from which the data is to be deleted. The WHERE clause filters which rows are to be deleted. In this example, only customer 10006 will be deleted. If the WHERE clause were omitted, this statement would delete every customer in the table.

DELETE takes no column names or wildcard characters. DELETE deletes entire rows, not columns. To delete specific columns, you use an UPDATE statement (as shown earlier in this chapter).

Note

Table Contents, Not Tables The DELETE statement deletes rows from tables; it can even delete all rows from a table. But DELETE never deletes the table itself.

Tip

Faster Deletes If you really do want to delete all rows from a table, don't use DELETE. Instead, use the TRUNCATE TABLE statement, which accomplishes the same thing but does it much more quickly. TRUNCATE actually drops and re-creates the table instead of deleting each row individually.

Guidelines for Updating and Deleting Data

The UPDATE and DELETE statements used in the previous sections all have WHERE clauses, and there is a very good reason for this. If you omit the WHERE clause, the UPDATE or DELETE is applied to every row in the table. In other words, if you execute an UPDATE without a WHERE clause, every row in the table is updated with the new values. Similarly, if you execute DELETE without a WHERE clause, all the contents of the table are deleted.

Here are some best practices that many SQL programmers follow:

- Never execute an UPDATE or a DELETE without a WHERE clause unless you really do intend to update or delete every row.
- Make sure every table has a primary key and use it as the WHERE clause whenever possible. You can specify individual primary keys, multiple values, or value ranges. (Refer to Chapter 15, "Joining Tables," if you have forgotten what a primary key is.)
- Before you use a WHERE clause with UPDATE or DELETE, first test it with a SELECT to make sure it is filtering the right records. It is far too easy to write incorrect WHERE clauses.
- Use database-enforced referential integrity (refer to Chapter 15 for this one, too) so MySQL will not allow the deletion of rows that have data in other tables related to them.

Caution

Use with Caution The bottom line is that MySQL has no Undo button. Be very careful when using UPDATE and DELETE, or you'll find yourself updating and deleting the wrong data.

Summary

In this chapter, you learned how to use the `UPDATE` and `DELETE` statements to manipulate the data in your tables. You learned the syntax for each of these statements, as well as the danger you face in using them. You also learned why `WHERE` clauses are so important in `UPDATE` and `DELETE` statements, and you were given guidelines to follow to help ensure that data does not get damaged inadvertently.

Challenges

1. In the United States, state name abbreviations should always be in uppercase. Write a SQL statement to update all U.S. addresses in both vendor states (`vend_state` in `Vendors`) and customer states (`cust_state` in `Customers`) so that they are uppercase. To do this, you'll need to use a function that converts text to uppercase (refer to Chapter 11, if needed) and a `WHERE` clause to filter just U.S. addresses.
2. Challenge 1 in Chapter 15 asked you to add yourself to the `Customers` table. Now delete yourself. Make sure to use a `WHERE` clause (and test it with a `SELECT` before using it in `DELETE`), or you'll delete all customers!

This page intentionally left blank

21

Creating and Manipulating Tables

In this chapter, you'll learn the basics of table creation, alteration, and deletion.

Creating Tables

MySQL statements are not used just for table data manipulation. Indeed, MySQL can be used to perform all database and table operations, including to create and manipulate tables.

There are generally two ways to create database tables:

- You can use an administration tool (like the ones discussed in Chapter 2, "Introducing MySQL") to create and manage database tables interactively.
- You can manipulate tables directly with MySQL statements.

To create tables programmatically, you use the `CREATE TABLE` SQL statement. It is worth noting that when you use interactive tools, you are actually using MySQL statements. Instead of writing these statements yourself, however, you have the MySQL Workbench interface generate and execute the MySQL for you.

Note

Additional Examples For additional examples of table creation scripts, see the code used to create the sample tables used in this book, as explained in Appendix B.

Basic Table Creation

To create a table using `CREATE TABLE`, you must specify the following information:

- The name of the new table, which you specify after the `CREATE TABLE` statement
- The name and definition of the table columns, separated by commas

The CREATE TABLE statement can also include other keywords and options, but at a minimum, you need the table name and column details. The following MySQL statement creates the customers table used throughout this book:

Input

```
CREATE TABLE customers
(
  cust_id      int       NOT NULL AUTO_INCREMENT,
  cust_name    char(50)  NOT NULL ,
  cust_address char(50)  NULL ,
  cust_city    char(50)  NULL ,
  cust_state   char(5)   NULL ,
  cust_zip     char(10)  NULL ,
  cust_country char(50)  NULL ,
  cust_contact char(50)  NULL ,
  cust_email   char(255) NULL ,
  PRIMARY KEY (cust_id)
) ENGINE=InnoDB;
```

Analysis

As you can see in this statement, the table name is specified immediately following the CREATE TABLE statement. The actual table definition (all the columns) is enclosed within parentheses. The columns are separated by commas. This particular table is made up of nine columns. Each column definition starts with the column name (which must be unique within the table), followed by the column's datatype. (Refer to Chapter 1, "Understanding SQL," for an explanation of datatypes. In addition, Appendix D, "MySQL Datatypes," lists the datatypes supported by MySQL.) The table's primary key may be specified at table creation time using the PRIMARY KEY keyword. Here, the column cust_id is specified as the primary key column. The entire statement is terminated with a semicolon after the closing parenthesis. (Ignore the ENGINE=InnoDB and AUTO_INCREMENT statements for now; we'll come back to them later.)

Tip

Statement Formatting As you will recall, white space is ignored in MySQL statements. You can type a statement on one long line or break it up over many lines. It makes no difference which way you do it. This flexibility enables you to format your SQL as best suits you. The preceding CREATE TABLE statement is a good example of MySQL statement formatting; the code is specified over multiple lines, with the column definitions indented for ease of reading and editing. Formatting your MySQL in this way is entirely optional but highly recommended.

> **Tip**
>
> **Handling Existing Tables** When you create a new table, the table name specified must not already exist, or you'll generate an error. To prevent accidental overwriting, SQL requires that you first manually remove a table (see later sections for details) and then re-create it rather than just overwrite it.
>
> If you want to be sure you're creating a table that does not already exist, specify `IF NOT EXISTS` after the table name. MySQL does not check to see that the schema of the existing table matches the one you are about to create. It simply checks to see if the table name exists, and it proceeds with table creation only if it does not.

Working with NULL Values

Back in Chapter 6, "Filtering Data," you learned that a `NULL` value is no value or the lack of a value. A column that allows `NULL` values also allows rows to be inserted with no value at all in that column. A column that does not allow `NULL` values does not accept rows with no value; in other words, that column will always be required when rows are inserted or updated.

Every table column is either a `NULL` column or a `NOT NULL` column, and that state is specified in the table definition at creation time. Take a look at the following example:

Input

```
CREATE TABLE orders
(
  order_num  int      NOT NULL AUTO_INCREMENT,
  order_date datetime NOT NULL ,
  cust_id    int      NOT NULL ,
  PRIMARY KEY (order_num)
) ENGINE=InnoDB;
```

Analysis

This statement creates the `orders` table used throughout this book. `orders` contains three columns: one for the order number, one for the order date, and one for the customer ID. All three columns are required, and so each contains the keyword `NOT NULL` to prevent the insertion of columns with no value. If someone tries to insert no value, an error will be returned, and the insertion will fail.

This next example creates a table with a mixture of `NULL` and `NOT NULL` columns:

Input

```
CREATE TABLE vendors
(
  vend_id      int NOT  NULL AUTO_INCREMENT,
  vend_name    char(50) NOT NULL ,
  vend_address char(50) NULL ,
  vend_city    char(50) NULL ,
```

```
  vend_state   char(5)  NULL ,
  vend_zip     char(10) NULL ,
  vend_country char(50) NULL ,
  PRIMARY KEY (vend_id)
) ENGINE=InnoDB;
```

► Analysis

This statement creates the `vendors` table used throughout this book. The vendor ID and vendor name columns are both required, and are, therefore, specified as `NOT NULL`. The five remaining columns all allow `NULL` values, and so `NOT NULL` is not specified. `NULL` is the default setting, so if `NOT NULL` is not specified, `NULL` is assumed.

Caution

Understanding `NULL` Don't confuse `NULL` values with empty strings. A `NULL` value is the lack of a value; it is not an empty string. You could, for example, specify `''` (two single quotes with nothing in between them) in a `NOT NULL` column because an empty string is a valid value; it is not no value. `NULL` values are specified with the keyword `NULL`, not with an empty string.

Primary Keys Revisited

As explained earlier, primary key values must be unique. That is, every row in a table must have a unique primary key value. If a single column is used for the primary key, it must be unique; if multiple columns are used, the combination of those columns must be unique.

The `CREATE TABLE` examples shown thus far use a single column as the primary key. The primary key is defined using a statement such as:

```
PRIMARY KEY (vend_id)
```

To create a primary key made up of multiple columns, simply specify the column names as a comma-delimited list, as shown in this example:

```
CREATE TABLE orderitems
(
  order_num  int          NOT NULL ,
  order_item int          NOT NULL ,
  prod_id    char(10)     NOT NULL ,
  quantity   int          NOT NULL ,
  item_price decimal(8,2) NOT NULL ,
  PRIMARY KEY (order_num, order_item)
) ENGINE=InnoDB;
```

The `orderitems` table contains the order specifics for each order in the `orders` table. There may be multiple items per order, but each order will only ever have one first item, one second item, and so on. As such, the combination of order number

(column order_num) and order item (column order_item) are unique, and thus the combination is suitable to be the primary key, which is defined as follows:

```
PRIMARY KEY (order_num, order_item)
```

Primary keys may be defined at table creation time (as shown here) or after table creation (as discussed later in this chapter).

Tip

No Defining Primary Keys with NULL Values Back in Chapter 1, you learned that primary keys are columns whose values uniquely identify every row in a table. Only columns that do not allow NULL values can be used in primary keys. Columns that allow no value at all cannot be used as unique identifiers.

Using AUTO_INCREMENT

Let's take a look at the customers and orders tables again. Customers in the customers table are uniquely identified by the column cust_id, which includes a unique number for each and every customer. Similarly, orders in the orders table each have a unique order number, which is stored in the column order_num. These numbers have no special significance other than the fact that they are unique. When a new customer or order is added, a new customer ID or order number is needed. The numbers can be anything, as long as they are unique.

Obviously, the simplest number to use would be whatever comes next—whatever is one higher than the current highest number. For example, if the highest cust_id is 10005, the next customer inserted into the table could have the cust_id 10006.

Simple, right? Well, not really. How would you determine the next number to be used? You could, of course, use a SELECT statement to get the highest number (using the Max() function introduced in Chapter 12, "Summarizing Data") and then add 1 to it. But that would not be safe, as you'd need to find a way to ensure that no one else inserted a row in between the time that you performed the SELECT and the INSERT, which is a legitimate possibility in multiuser applications. It also wouldn't be efficient (as performing additional MySQL operations is never ideal).

This is where AUTO_INCREMENT comes in. Look at the following line, which is part of the CREATE TABLE statement used to create the customers table:

```
cust_id      int       NOT NULL AUTO_INCREMENT
```

AUTO_INCREMENT tells MySQL that this column is to be automatically incremented each time a row is added. Each time an INSERT operation is performed, MySQL automatically increments (hence the name AUTO_INCREMENT) the column, assigning it the next available value. This way, each row is assigned a unique cust_id, which is then used as the primary key value.

Only one AUTO_INCREMENT column is allowed per table, and it must be indexed (for example, by being made a primary key).

Note

Overriding `AUTO_INCREMENT` Need to use a specific value in a column designated as `AUTO_INCREMENT`? You can. Simply specify a value in the `INSERT` statement, and as long as it is unique (that is, has not been used yet), that value will be used instead of an automatically generated one. Subsequent incrementing will start using the value manually inserted. (See the table population scripts in Appendix B for examples of this.)

Tip

Determining the `AUTO_INCREMENT` Value One downside of having MySQL generate primary keys for you (via `AUTO_INCREMENT`) is that you don't know what those values are.

Consider this scenario: You are adding a new order. This requires creating a single row in the `orders` table and then a row for each item ordered in the `orderitems` table. The order number is stored, along with the order details, in `orderitems`. This is how the `orders` and `orderitems` table are related to each other. And this obviously requires that you know the generated `order_num` after the order's row is inserted and before the `orderitems` rows are inserted.

So how could you obtain this value when an `AUTO_INCREMENT` column is used? By using the `last_insert_id()` function, like this:

```
SELECT last_insert_id();
```

This returns the last `AUTO_INCREMENT` value, which you can then use in subsequent MySQL statements.

Specifying Default Values

MySQL enables you to specify default values to be used if no value is specified when a row is inserted. Default values are specified using the `DEFAULT` keyword in the column definitions in the `CREATE TABLE` statement.

Look at the following example:

Input

```
CREATE TABLE orderitems
(
  order_num  int          NOT NULL ,
  order_item int          NOT NULL ,
  prod_id    char(10)     NOT NULL ,
  quantity   int          NOT NULL DEFAULT 1,
  item_price decimal(8,2) NOT NULL ,
  PRIMARY KEY (order_num, order_item)
) ENGINE=InnoDB;
```

Analysis

This statement creates the `orderitems` table, which contains the individual items that make up an order. (The order itself is stored in the `orders` table.) The `quantity` column contains the quantity for each item in an order. In this example, adding the text `DEFAULT 1` to the column description instructs MySQL to use a quantity of 1 if no quantity is specified.

Caution

Functions Are Not Allowed Unlike most DBMSs, MySQL does not allow the use of functions as `DEFAULT` values; only constants are supported.

Tip

Using `DEFAULT` Instead of `NULL` Values Many database developers use `DEFAULT` values instead of `NULL` columns, especially in columns that will be used in calculations or data groupings.

Engine Types

You might have noticed that the `CREATE TABLE` statements used thus far have all ended with an `ENGINE=InnoDB` statement.

Like every other DBMS, MySQL has an internal engine that actually manages and manipulates data. When you use the `CREATE TABLE` statement, that internal engine is used to actually create the tables, and when you use the `SELECT` statement or perform any other database processing, the engine is used to process your request. For the most part, the engine is buried within the DBMS, and you need not pay much attention to it.

But unlike every other DBMS, MySQL does not come with a single engine. Rather, it ships with several engines, all buried within the MySQL server and all capable of executing commands such as `CREATE TABLE` and `SELECT`.

So why bother shipping multiple engines? Because they each have different capabilities and features, and being able to pick the right engine for the job gives you unprecedented power and flexibility.

Of course, you are free to totally ignore database engines. If you omit the `ENGINE=` statement, the default engine is used (most likely MyISAM), and most of your SQL statements will work as is. But not all of them will, and that is why this is important (and why two engines are used in the sample tables used in this book).

Here are several engines to be aware of:

- InnoDB is a transaction-safe engine (see Chapter 26, "Managing Transaction Processing"). It does not support full-text searching.
- MEMORY is functionally equivalent to MyISAM, but data is stored in memory (instead of on disk), so it is extremely fast (and ideally suited for temporary tables).
- MyISAM is a very high-performance engine. It supports full-text searching (see Chapter 18, "Full-Text Searching") but does not support transactional processing.

Note

To Learn More For a complete list of supported engines, see the MySQL documentation.

Engine types may be mixed. The example tables used throughout this book all use InnoDB with the exception of the `productnotes` table, which uses MyISAM. The reason for this is that I wanted support for transactional processing (and thus used InnoDB) but also needed full-text searching support in `productnotes` (and thus used MyISAM for that table).

Caution

Foreign Keys Can't Span Engines There is one big downside to mixing engine types. Foreign keys (used to enforce referential integrity, as explained in Chapter 1) cannot span engines. That is, a table using one engine cannot have a foreign key that refers to a table that uses another engine.

So, which engine should you use? Well, it depends on what features you need. MyISAM tends to be the most popular engine because of its performance and features. But if you do need transaction-safe processing, you will need to use a different engine.

Updating Tables

To update table definitions, you use the `ALTER TABLE` statement. However, ideally, the design of a table should never be altered once the table contains data. You should spend sufficient time anticipating future needs during the table design process so that extensive changes are not required later on.

When you simply must change a table, you can do so by using `ALTER TABLE`. You must specify the following information with it:

- The name of the table to be altered after the keywords `ALTER TABLE` (The table you specify must exist, or an error will be generated.)
- The list of changes to be made

The following example adds a column to a table:

Input

```
ALTER TABLE vendors
ADD vend_phone CHAR(20);
```

Analysis

This statement adds a column named `vend_phone` to the `vendors` table. The datatype must be specified.

To remove this newly added column, you can use the following:

Input

```
ALTER TABLE Vendors
DROP COLUMN vend_phone;
```

One common use for `ALTER TABLE` is to define foreign keys. The following is the code used to define the foreign keys used by the tables in this book:

```
ALTER TABLE orderitems
ADD CONSTRAINT fk_orderitems_orders
FOREIGN KEY (order_num) REFERENCES orders (order_num);

ALTER TABLE orderitems
ADD CONSTRAINT fk_orderitems_products FOREIGN KEY (prod_id) REFERENCES products
  (prod_id);

ALTER TABLE orders
ADD CONSTRAINT fk_orders_customers FOREIGN KEY (cust_id) REFERENCES customers
  (cust_id);

ALTER TABLE products
ADD CONSTRAINT fk_products_vendors
FOREIGN KEY (vend_id) REFERENCES vendors (vend_id);
```

Four `ALTER TABLE` statements are used here because four different tables are being altered. To make multiple alterations to a single table, you can use a single `ALTER TABLE` statement and list the alterations, separated by commas.

Complex table structure changes usually require a manual move process involving these steps:

1. Create a new table with the new column layout.
2. Use the `INSERT SELECT` statement to copy the data from the old table to the new table. (See Chapter 19, "Inserting Data," for details on the `INSERT SELECT` statement.) Use conversion functions and calculated fields, if needed.
3. Verify that the new table contains the desired data.
4. Rename the old table (or delete it, if you are really brave).
5. Rename the new table with the name previously used by the old table.
6. Re-create any triggers, stored procedures, indexes, and foreign keys, as needed.

Caution

Use `ALTER TABLE` Carefully Use `ALTER TABLE` with extreme caution and be sure you have a complete set of backups (both schema and data) before proceeding. Database table changes cannot be undone—and if you add columns you don't need, you might not be able to remove them. Similarly, if you drop a column that you do need, you might lose all the data in that column.

Deleting Tables

Deleting a table (that is, actually removing the entire table and not just the contents) is very easy—arguably too easy. You deleted a table by using the `DROP TABLE` statement:

Input

```
DROP TABLE customers2;
```

Analysis

This statement deletes the `customers2` table (assuming that it exists). You get no confirmation, and there is no undo. When you execute this statement, MySQL permanently removes the table.

Renaming Tables

To rename a table, use the `RENAME TABLE` statement as follows:

Input

```
RENAME TABLE customers2 TO customers;
```

Analysis

`RENAME TABLE` does just what it says it does: It renames a table. You can rename multiple tables in one operation like this:

```
RENAME TABLE backup_customers TO customers,
             backup_vendors TO vendors,
             backup_products TO products;
```

Summary

In this chapter, you learned several new SQL statements. `CREATE TABLE` is used to create new tables, `ALTER TABLE` is used to change table columns (or other objects, like constraints or indexes), and `DROP TABLE` is used to completely delete a table. These statements should be used with extreme caution—and only after backups have been made. You also learned about database engines, defining primary and foreign keys, and other important table and column options.

Challenges

1. Add a website column (`vend_web`) to the `Vendors` table. You need a text field big enough to accommodate a URL.
2. Use `UPDATE` statements to update `Vendors` table records to include a website. (You can make up any address.)

22

Using Views

In this chapter, you'll learn exactly what views are, how they work, and when they should be used. You'll also see how views can be used to simplify some of the SQL operations performed in earlier chapters.

Understanding Views

Note

This Chapter Requires MySQL 5 or Later Support for views was added to MySQL 5. Therefore, this chapter is applicable to MySQL 5 or later only.

Views are virtual tables. Unlike tables that contain data, views simply contain queries that dynamically retrieve data when used.

The best way to understand views is to look at an example. Back in Chapter 15, "Joining Tables," you used the following `SELECT` statement to retrieve data from three tables:

Input

```
SELECT cust_name, cust_contact
FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
  AND orderitems.order_num = orders.order_num
  AND prod_id = 'TNT2';
```

That query was used to retrieve the customers who had ordered a specific product. Anyone needing this data would have to understand the table structure, as well as how to create the query and join the tables. To retrieve the same data for another product (or for multiple products), the tables would all need to be joined, and the last `WHERE` clause would have to be modified.

Now imagine that you could wrap that entire query, complete with all the table relationships defined, in a virtual table called `productcustomers`. You could then simply use the following to retrieve the same data:

Input

```
SELECT cust_name, cust_contact
FROM productcustomers
WHERE prod_id = 'TNT2';
```

This is where views come into play. `productcustomers` is a view, and as a view, it does not contain any actual columns or data, as a table would. Instead, it contains a SQL query—the same query used previously to join the tables.

Why Use Views

You've already seen one use for views. Here are some other common uses:

- To reuse SQL statements.
- To simplify complex SQL operations. After a query is written, it can be reused easily, without requiring the user to know the details of the underlying query.
- To expose parts of a table instead of the complete table.
- To secure data. Users can be given access to specific subsets of tables instead of to entire tables.
- To change data formatting and representation. Views can return data formatted and presented differently than in their underlying tables.

For the most part, after views are created, they can be used in the same way as tables. You can perform `SELECT` operations, filter and sort data, join views to other views or tables, and possibly even add and update data. (There are some restrictions on this last item. More on that in a moment.)

The important thing to remember is that views provide views into data stored elsewhere. Views contain no data themselves, and the data they return is retrieved from other tables. When data is added or changed in those tables, the views return the changed data.

Caution

Performance Issues Because views contain no data, any retrieval needed to execute a query must be processed every time the view is used. If you create complex views with multiple joins and filters, or if you nest views, you may find that performance is dramatically degraded. Be sure you test execution before deploying applications that use views extensively.

View Rules and Restrictions

Here are some of the most common rules and restrictions governing view creation and usage:

- Like tables, views must be uniquely named. (That is, a view cannot be given the same name as any other table or view.)
- There is no limit to the number of views that can be created.
- To create views, you must have security access. This is usually granted by the database administrator.
- Views can be nested; that is, a view may be built using a query that retrieves data from another view.
- `ORDER BY` can be used in a view, but it will be overridden if `ORDER BY` is also used in the `SELECT` that retrieves data from the view.
- Views cannot be indexed, and they cannot have triggers or default values associated with them.
- Views can be used in conjunction with tables (for example, to create a `SELECT` statement that joins a table and a view).

Using Views

Now that you know what views are (and the rules and restrictions that govern them), let's look at view creation:

- Views are created using the `CREATE VIEW` statement.
- To view the statement used to create a view, use `SHOW CREATE VIEW` *`viewname`*`;`.
- To remove a view, use the `DROP` statement. The syntax is simply `DROP VIEW` *`viewname`*`;`.
- To update a view, you can use the `DROP` statement and then the `CREATE` statement again, or you can just use `CREATE OR REPLACE VIEW`, which creates the view if it does not exist and replaces it if it does.

Using Views to Simplify Complex Joins

One of the most common uses of views is to hide complex SQL, and this often involves joins. Look at the following statement:

Input

```
CREATE VIEW productcustomers AS
SELECT cust_name, cust_contact, prod_id
FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
  AND orderitems.order_num = orders.order_num;
```

Analysis

This statement creates a view named `productcustomers`, which joins three tables to return a list of all customers who have ordered any product. If you use `SELECT * FROM productcustomers`, you get a list of every customer who has ordered anything.

To retrieve a list of customers who have ordered product TNT2, you can use the following:

Input

```
SELECT cust_name, cust_contact
FROM productcustomers
WHERE prod_id = 'TNT2';
```

Output

```
+----------------+-----------------+
| cust_name      |  cust_contact   |
+----------------+-----------------+
| Coyote Inc.    | Y Lee           |
| Yosemite Place | Y Sam           |
+----------------+-----------------+
```

Analysis

This statement retrieves specific data from the view by issuing a `WHERE` clause. When MySQL processes the request, it adds the specified `WHERE` clause to any existing `WHERE` clauses in the view query so the data is filtered correctly.

As you can see, views can greatly simplify the use of complex SQL statements. By using a view, you can write the underlying SQL once and then reuse it as needed.

Tip

Creating Reusable Views It is a good idea to create views that are not tied to specific data. For example, the view created in this example returns customers for all products, not just product TNT2 (for which the view was first created). By expanding the scope of a view, you enable it to be reused, making it even more useful. You also eliminate the need to create and maintain multiple similar views.

Using Views to Reformat Retrieved Data

As mentioned previously, another common use of views is for reformatting retrieved data. The following `SELECT` statement (from Chapter 10, "Creating Calculated Fields") returns the vendor name and location in a single combined calculated column:

Input

```
SELECT Concat(RTrim(vend_name),
              ' (', RTrim(vend_country), ')')
       AS vend_title
FROM vendors
ORDER BY vend_name;
```

Output

```
+------------------------+
| vend_title             |
+------------------------+
| ACME (USA)             |
| Anvils R Us (USA)      |
| Furball Inc. (USA)     |
| Jet Set (England)      |
| Jouets Et Ours (France)|
| LT Supplies (USA)      |
+------------------------+
```

Now suppose that you regularly need results in this format. Rather than perform the concatenation each time it is needed, you can create a view and use that instead. To turn this statement into a view, you can use the following:

Input

```
CREATE VIEW vendorlocations AS
SELECT Concat(RTrim(vend_name),
       ' (', RTrim(vend_country), ')')
       AS vend_title
FROM vendors
ORDER BY vend_name;
```

Analysis

This statement creates a view using exactly the same query as the previous `SELECT` statement. To retrieve the data to create all mailing labels, simply use the following:

Input

```
SELECT *
FROM vendorlocations;
```

Output

```
+------------------------+
| vend_title             |
+------------------------+
| ACME (USA)             |
| Anvils R Us (USA)      |
| Furball Inc. (USA)     |
| Jet Set (England)      |
| Jouets Et Ours (France)|
| LT Supplies (USA)      |
+------------------------+
```

Using Views to Filter Unwanted Data

Views are also useful for applying common `WHERE` clauses. For example, you might want to define a `customeremaillist` view so it filters out customers without email addresses. To do this, you can use the following statement:

Input

```
CREATE VIEW customeremaillist AS
SELECT cust_id, cust_name, cust_email
FROM customers
WHERE cust_email IS NOT NULL;
```

Analysis

Obviously, when sending email to a mailing list, you'd want to ignore users who have no email address. The `WHERE` clause here filters out rows that have `NULL` values in the `cust_email` columns so they will not be retrieved.

You can now use the `customeremaillist` view for data retrieval just as you would any table. Consider this example:

Input

```
SELECT *
FROM customeremaillist;
```

Output

```
+---------+---------------+-----------------------+
| cust_id | cust_name     | cust_email            |
+---------+---------------+-----------------------+
|   10001 | Coyote Inc.   | ylee@coyote.com       |
|   10003 | Wascals       | rabbit@wascally.com   |
|   10004 | Yosemite Place | sam@yosemite.com     |
+---------+---------------+-----------------------+
```

Note

`WHERE` Clauses If a `WHERE` clause is used when retrieving data from a view, the two sets of clauses (the one in the view and the one passed to it) will be combined automatically.

Using Views with Calculated Fields

Views are exceptionally useful for simplifying the use of calculated fields. The following is a `SELECT` statement introduced in Chapter 10 that retrieves the order items for a specific order and calculates the expanded price for each item:

Input

```
SELECT prod_id,
       quantity,
       item_price,
```

```
       quantity*item_price AS expanded_price
FROM orderitems
WHERE order_num = 20005;
```

► Output

```
+---------+----------+------------+----------------+
| prod_id | quantity | item_price | expanded_price |
+---------+----------+------------+----------------+
| ANV01   |       10 | 5.99       | 59.90          |
| ANV02   |        3 | 9.99       | 29.97          |
| TNT2    |        5 | 10.00      | 50.00          |
| FB      |        1 | 10.00      | 10.00          |
+---------+----------+------------+----------------+
```

To turn this into a view, use the following:

► Input

```
CREATE VIEW orderitemsexpanded AS
SELECT order_num,
       prod_id, quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM orderitems;
```

To retrieve the details for order 20005 (the previous output), use the following:

► Input

```
SELECT *
FROM orderitemsexpanded
WHERE order_num = 20005;
```

► Output

```
+-----------+---------+----------+------------+----------------+
| order_num | prod_id | quantity | item_price | expanded_price |
+-----------+---------+----------+------------+----------------+
|   20005   | ANV01   |       10 | 5.99       | 59.90          |
|   20005   | ANV02   |        3 | 9.99       | 29.97          |
|   20005   | TNT2    |        5 | 10.00      | 50.00          |
|   20005   | FB      |        1 | 10.00      | 10.00          |
+-----------+---------+----------+------------+----------------+
```

As you can see, views are easy to create and even easier to use. When used correctly, views can greatly simplify complex data manipulation.

Updating Views

All of the views thus far have been used with `SELECT` statements. But can view data be updated? It depends.

As a rule, yes, views are updatable (that is, you can use `INSERT`, `UPDATE`, and `DELETE` on them). Updating a view updates the underlying table. (Recall that the view has no data

of its own.) If you add or remove rows from a view you are actually removing them from the underlying table.

But not all views are updatable. Basically, if MySQL is unable to correctly ascertain the underlying data to be updated, updates (including insertions and deletions) are not allowed. In practice, this means that if any of the following are used, you will not be able to update the view:

- Grouping (using `GROUP BY` and `HAVING`)
- Joins
- Subqueries
- Unions
- Aggregate functions (`Min()`, `Count()`, `Sum()`, and so forth)
- `DISTINCT`
- Derived (calculated) columns

In other words, many of the examples used in this chapter would not be updatable. This might sound like a serious restriction, but it really isn't because views are primarily used for data retrieval.

Tip

Use Views for Retrieval As a rule, you should use views for data retrieval (`SELECT` statements) and not for updates (`INSERT`, `UPDATE`, and `DELETE` statements).

Summary

Views are virtual tables. They do not contain data; rather, they contain queries that retrieve data as needed. Views provide a level of encapsulation around MySQL `SELECT` statements and can be used to simplify data manipulation as well as to reformat or secure underlying data.

Challenges

1. Create a view called `VendorProducts` that joins `Vendors` and `Products` tables. Use a `SELECT` to make sure you have the right data.
2. Create a view called `CustomersWithOrders` that contains all of the columns in `Customers` but includes only customers who have placed orders. Here's a hint: You can use `JOIN` on the `Orders` table to filter just the customers you want. Then use `SELECT` to make sure you have the right data.

23

Working with Stored Procedures

In this chapter, you'll learn what stored procedures are, why they are used, and how they are used. You'll also learn the basic syntax for creating and using them.

Understanding Stored Procedures

> **Note**
> **This Chapter Requires MySQL 5** Support for stored procedures was added to MySQL 5. Therefore, this chapter is applicable to MySQL 5 or later only.

Most of the SQL statements that we've used thus far are simple in that they use a single statement against one or more tables. Not all operations are that simple, though. Often, multiple statements are needed to perform a complete operation. For example, consider the following scenario:

- To process an order, checks must be made to ensure that items are in stock.
- If items are in stock, they need to be reserved so they are not sold to anyone else, and the available quantity must be reduced to reflect the correct amount in stock.
- Any items not in stock need to be ordered; this requires some interaction with the vendor.
- The customer needs to be notified about which items are in stock (and can be shipped immediately) and which are back-ordered.

This is obviously not a complete example, and it is even beyond the scope of the example tables that we have been using in this book, but it will suffice to help make a point. Performing this process requires using many MySQL statements against many tables. In addition, the exact statements that need to be performed and their order are not fixed; they can (and will) vary according to which items are in stock and which are not.

How would you write this code? You could write each of the statements individually and execute other statements conditionally, based on the result. You'd have to do this every time this processing was needed (and in every application that needed it).

Or you could create a stored procedure. A stored procedure is simply a collection of one or more MySQL statements saved for future use. You can think of stored procedures as batch files, although in truth they are more than that.

Why Use Stored Procedures

Now that you know what stored procedures are, why use them? There are many reasons, but here are the primary ones:

- They simplify complex operations (as illustrated in the previous example) by encapsulating processes into a single easy-to-use unit.
- They ensure data integrity by not requiring that a series of steps be created over and over. If all developers and applications use the same (tried and tested) stored procedure, the same code will be used by all.
  An extension of this is to prevent errors. The more steps that need to be performed, the more likely it is that errors will be introduced. Preventing errors ensures data consistency.
- They simplify change management. If tables, column names, or business logic (or just about anything else) changes, only the stored procedure code needs to be updated, and no one else needs to even be aware that changes were made.
  An extension of this is security. Restricting access to underlying data via stored procedures reduces the chance of data corruption (unintentional or otherwise).
- They improve performance, as they typically execute more quickly than do individual SQL statements.
- There are MySQL language elements and features that are available only within single requests. Stored procedures can use them to write code that is more powerful and flexible. (We'll see an example of this in the next chapter.)

In other words, there are three primary benefits to stored procedures: simplicity, security, and performance. Obviously, all these factors are extremely important. Before you run off and turn all your SQL code into stored procedures, though, consider the downsides:

- Stored procedures tend to be more complex to write than basic SQL statements, and writing them requires a greater degree of skill and experience.
- You might not have the security access needed to create stored procedures. Many database administrators restrict stored procedure creation rights, allowing users to execute them but not necessarily create them.

Nonetheless, stored procedures are very useful and should be used whenever possible.

> **Note**
> **Can't Write Them? You Can Still Use Them** MySQL distinguishes the security and access needed to write stored procedures from the security and access needed to execute them. This is a good thing; even if you can't (or don't want to) write your own stored procedures, you can still execute them, when appropriate.

Using Stored Procedures

Using stored procedures requires knowing how to execute (that is, run) them. Stored procedures are executed far more often than they are written, so we'll start there. And then we'll look at creating and working with stored procedures.

Executing Stored Procedures

MySQL refers to executing a stored procedure as *calling* the procedure, and the MySQL statement to execute a stored procedure is `CALL`. `CALL` takes the name of the stored procedure and any parameters that need to be passed to it. Take a look at this example:

► Input

```
CALL productpricing(@pricelow,
                    @pricehigh,
                    @priceaverage);
```

► Analysis

Here a stored procedure named `productpricing` is executed. It calculates and returns the lowest, highest, and average product prices. (And, no, you can't run this example just yet; stay tuned.)

Stored procedures might or might not display results, as you will see shortly.

Creating Stored Procedures

As already explained, writing a stored procedure is not trivial. To give you a taste for what is involved, let's look at a simple example of a stored procedure that returns the average product price. Here is the code:

► Input

```
DELIMITER //

CREATE PROCEDURE productpricing()
BEGIN
   SELECT Avg(prod_price) AS priceaverage
   FROM products;
END//

DELIMITER ;
```

► Analysis

Ignore the first and last lines for a moment; we'll come back to them shortly. The stored procedure is named `productpricing` and is defined with the statement `CREATE PROCEDURE productpricing()`. If the stored procedure accepted parameters, they would be enumerated between the `(` and `)`. This stored procedure has no parameters, but the trailing `()` is still required. `BEGIN` and `END` statements are used to delimit the body of the stored procedure, and the body in this case is just a simple `SELECT` statement (using the `Avg()` function you learned about in Chapter 12, "Summarizing Data").

When MySQL processes this code, it creates a new stored procedure named `productpricing`. No data is returned because the code does not call the stored procedure; it simply creates the stored procedure for future use.

The DELIMITER Challenge

Back to the first and last lines in the example you just saw. As you have repeatedly seen, MySQL relies on the `;` character to terminate SQL statements. The `;` character is called the *delimiter* because it delimits (that is, defines boundaries) between SQL statements. That creates a problem in our statement. Look at this code:

```
CREATE PROCEDURE productpricing()
BEGIN
   SELECT Avg(prod_price) AS priceaverage
   FROM products;
END;
```

The closing `END;` terminates the `CREATE` statement. But look carefully, and you see that there's another `;` in there, after `products`, and that `;` terminates the statement prematurely, before the closing `END`.

The solution is to temporarily change the command-line utility delimiter, as shown here:

```
DELIMITER //
```

`DELIMITER //` instructs MySQL to use `//` instead of `;` as the new end-of-statement delimiter. And indeed, the `END` that closes the stored procedure is defined as `END //` instead of the expected `END;`. This way, the `;` within the stored procedure body remains intact and is correctly passed to the database engine. And then, things are restored back to how they were initially, like this:

```
DELIMITER ;
```

Note

Doesn't Have to Be ; Any character may be used as the delimiter except for `\`. Just be sure to use something that doesn't have a special meaning in SQL.

So how would you use this stored procedure? Like this:

Input

```
CALL productpricing();
```

Output

```
+--------------+
| priceaverage |
+--------------+
|    16.133571 |
+--------------+
```

Analysis

`CALL productpricing();` executes the just-created stored procedure and displays the returned result. Because a stored procedure is actually a type of function, `()` characters are required after the stored procedure name (even when no parameters are being passed).

Dropping Stored Procedures

After they are created, stored procedures remain on the server, ready for use, until dropped. The `DROP` command (similar to the `DROP` statement you saw in Chapter 21, "Creating and Manipulating Tables") removes the stored procedure from the server.

To remove the stored procedure just created, use the following statement:

Input

```
DROP PROCEDURE productpricing;
```

Analysis

This removes the just-created stored procedure. Notice that the trailing `()` is not used; here just the stored procedure name is specified.

Tip

Drop Only if It Exists `DROP PROCEDURE` will throw an error if the named procedure does not actually exist. To delete a procedure if it exists (and not throw an error if it does not), use `DROP PROCEDURE IF EXISTS`.

Working with Parameters

`productpricing` is a really simple stored procedure. It simply displays the results of a `SELECT` statement. Typically stored procedures do not display results; rather, they return them into variables that you specify.

New Term

Variable A named location in memory that is used to temporarily store data.

Here is an updated version of `productpricing` (but note that you won't be able to create the stored procedure again if you did not previously drop it):

Input

```
DELIMITER //

CREATE PROCEDURE productpricing(
   OUT pl DECIMAL(8,2),
   OUT ph DECIMAL(8,2),
   OUT pa DECIMAL(8,2)
)
BEGIN
   SELECT Min(prod_price)
   INTO pl
   FROM products;
   SELECT Max(prod_price)
   INTO ph
   FROM products;
   SELECT Avg(prod_price)
   INTO pa
   FROM products;
END//
DELIMITER ;
```

Analysis

This stored procedure accepts three parameters: `pl` to store the lowest product price, `ph` to store the highest product price, and `pa` to store the average product price (hence the variable names). Each parameter must have its type specified; here a decimal value is used. The keyword `OUT` is used to specify that this parameter is used to send a value out of the stored procedure (back to the caller).

MySQL supports parameters of types `IN` (those passed to stored procedures), `OUT` (those passed from stored procedures, as we've used here), and `INOUT` (those used to pass parameters to and from stored procedures). The stored procedure code itself is enclosed within `BEGIN` and `END` statements, as shown earlier, and a series of `SELECT` statements are used to retrieve values that are then saved into the appropriate variables (specified with the `INTO` keyword).

Note

Parameter Datatypes The datatypes allowed in stored procedure parameters are the same as the ones used in tables. Appendix D, "MySQL Datatypes," lists these types.

Note that a recordset is not an allowed type, and so multiple rows and columns could not be returned via a parameter. This is why three parameters (and three `SELECT` statements) are used in the previous example.

To call this updated stored procedure, three variable names must be specified, as shown here:

► Input

```
CALL productpricing(@pricelow,
                    @pricehigh,
                    @priceaverage);
```

► Analysis

As the stored procedure expects three parameters, exactly three parameters must be passed—no more and no less. Therefore, three parameters are passed to this `CALL` statement. These are the names of the three variables that the stored procedure will store the results in.

Note

Variable Names A MySQL variable name must begin with `@`.

When called, this statement does not actually display any data. Rather, it returns variables that can then be displayed (or used in other processing).

To display the retrieved average product price, you could use the following:

► Input

```
SELECT @priceaverage;
```

► Output

```
+---------------+
| @priceaverage |
+---------------+
| 16.133571428  |
+---------------+
```

To obtain all three values, you can use the following:

► Input

```
SELECT @pricehigh, @pricelow, @priceaverage;
```

► Output

```
+------------+-----------+---------------+
| @pricehigh | @pricelow | @priceaverage |
+------------+-----------+---------------+
| 55.00      | 2.50      | 16.133571428  |
+------------+-----------+---------------+
```

Here is another example, this one using both `IN` and `OUT` parameters. `ordertotal` accepts an order number and returns the total for that order:

► Input

```
DELIMITER //

CREATE PROCEDURE ordertotal(
   IN onumber INT,
   OUT ototal DECIMAL(8,2)
)
BEGIN
   SELECT Sum(item_price*quantity)
   FROM orderitems
   WHERE order_num = onumber
   INTO ototal;
END //

DELIMITER ;
```

► Analysis

`onumber` is defined as `IN` because the order number is passed into the stored procedure. `ototal` is defined as `OUT` because the total is to be returned from the stored procedure. The `SELECT` statement uses both of these parameters, the `WHERE` clause uses `onumber` to select the right rows, and `INTO` uses `ototal` to store the calculated total.

To invoke this new stored procedure, you can use the following:

► Input

```
CALL ordertotal(20005, @total);
```

► Analysis

Two parameters must be passed to `ordertotal`; the first is the order number, and the second is the name of the variable that will contain the calculated total.

To display the total, you can then use the following:

► Input

```
SELECT @total;
```

► Output

```
+--------+
| @total |
+--------+
| 149.87 |
+--------+
```

► Analysis

`@total` has already been populated by the `CALL` statement to `ordertotal`, and `SELECT` displays the value it contains.

To obtain a display for the total of another order, you would need to call the stored procedure again and then redisplay the variable:

Input

```
CALL ordertotal(20009, @total);
SELECT @total;
```

Building Intelligent Stored Procedures

All of the stored procedures used thus far have basically encapsulated simple MySQL `SELECT` statements. And while they are all valid examples of stored procedures, they really don't do anything more than what you could do with those statements directly. (If anything, they just make things a little more complex.) The real power of stored procedures is realized when business rules and intelligent processing are included within them.

Consider this scenario: You need to obtain order totals as before, and you also need to add sales tax to the total for some customers (perhaps the ones in your own state). Now you need to do several things:

- Obtain the total (as before).
- Conditionally add tax to the total.
- Return the total (with or without tax).

This a perfect job for a stored procedure:

Input

```
DELIMITER //

-- Name: ordertotal
-- Parameters: onumber = order number
--             taxable = 0 if not taxable, 1 if taxable
--             ototal  = order total variable

CREATE PROCEDURE ordertotal(
   IN onumber INT,
   IN taxable BOOLEAN,
   OUT ototal DECIMAL(8,2)
) COMMENT 'Obtain order total, optionally adding tax'
BEGIN

   -- Declare variable for total
   DECLARE total DECIMAL(8,2);
   -- Declare tax percentage
   DECLARE taxrate INT DEFAULT 6;

   -- Get the order total
   SELECT Sum(item_price*quantity)
   FROM orderitems
   WHERE order_num = onumber
   INTO total;
```

```
   -- Is this taxable?
   IF taxable THEN
      -- Yes, so add taxrate to the total
      SELECT total+(total/100*taxrate) INTO total;
   END IF;

   -- And finally, save to out variable
   SELECT total INTO ototal;

END //

DELIMITER ;
```

► Analysis

The stored procedure has changed dramatically. First of all, comments have been added throughout (preceded by `--`). Comments are extremely important as stored procedures increase in complexity. An additional parameter has been added here: `taxable` is a `BOOLEAN` (that is true if taxable and false if not). Within the stored procedure body, two local variables are defined using `DECLARE` statements. `DECLARE` requires that a variable name and datatype be specified, and it also supports optional default values. (`taxrate` in this example is set to 6%.) The `SELECT` statement has changed so the result is stored in `total` (the local variable) instead of `ototal`. Then an `IF` statement checks to see if `taxable` is true, and if it is, another `SELECT` statement is used to add the tax to the local variable `total`. Finally, `total` (which might or might not have had tax added) is saved to `ototal` using another `SELECT` statement.

Tip

The `COMMENT` Keyword The stored procedure in this example includes a `COMMENT` value in the `CREATE PROCEDURE` statement. This is not required, but if it is specified, it is displayed in `SHOW PROCEDURE STATUS` results.

This is obviously a more sophisticated and powerful stored procedure. It also demonstrates a really important use for stored procedures in that it encapsulates all of the details pertaining to table and data structures and related business logic. Users could call this stored procedure to get the data they need without needing to know all the details of how the calculations work.

Now let's test the new stored procedure by using the following two statements:

► Input

```
CALL ordertotal(20005, 0, @total);
SELECT @total;
```

► Output

```
+--------+
| @total |
+--------+
| 149.87 |
+--------+
```

Input

```
CALL ordertotal(20005, 1, @total);
SELECT @total;
```

Output

```
+---------------+
| @total        |
+---------------+
| 158.862200000 |
+---------------+
```

Analysis

We've added one parameter that specifies whether to calculate tax. `BOOLEAN` values may be specified as `1` for true and `0` for false. (Actually, any nonzero value is considered true, and only `0` is considered false.) By specifying `0` or `1` in the middle parameter, you can conditionally add tax to the order total.

Note

The `IF` Statement This example shows the basic use of the MySQL `IF` statement. `IF` also supports `ELSEIF` and `ELSE` clauses; the former also uses a `THEN` clause, and the latter does not. We'll be seeing additional uses of `IF` (as well as other flow control statements) in future chapters.

Inspecting Stored Procedures

To display the `CREATE` statement used to create a stored procedure, use the `SHOW CREATE PROCEDURE` statement:

Input

```
SHOW CREATE PROCEDURE ordertotal;
```

To obtain a list of stored procedures, including details on when and who created them, use `SHOW PROCEDURE STATUS`.

Note

Limiting Procedure Status Results `SHOW PROCEDURE STATUS` lists all stored procedures. To restrict the output, you can use `LIKE` to specify a filter pattern, as in this example:

```
SHOW PROCEDURE STATUS LIKE 'ordertotal';
```

Summary

In this chapter, you learned what stored procedures are and why they are used. You also learned the basics of stored procedure execution and creation syntax, and you saw some of the ways stored procedures can be used. We'll continue this subject when we look at cursors in the next chapter.

Challenges

1. Create a stored procedure that accepts a customer ID and returns all orders made by that customer.
2. Different locations have different tax rates. The `ordertotal` stored procedure hard-coded the tax rate as 6%. Update that stored procedure so that it accepts the tax rate to use, if one is needed. Here's a hint: You actually don't need another parameter but could replace the `taxable` flag to accept a tax rate, with `0` meaning no tax (yay!).

24

Using Cursors

In this chapter, you'll learn what cursors are and how to use them.

Understanding Cursors

Note
This Chapter Requires MySQL 5 Support for cursors was added to MySQL 5. Therefore, this chapter is applicable to MySQL 5 or later only.

As you have seen in previous chapters, MySQL retrieval operations work with sets of rows known as *result sets*. The rows returned are all the rows (zero or more of them) that match a SQL statement. Using simple `SELECT` statements, there is no way to get the first row, the next row, or the previous 10 rows, for example. There is also no way to process all rows one at a time (as opposed to processing all of them in a batch).

Sometimes there is a need to step through rows forward or backward and one or more at a time. This is what cursors are used for. A cursor is a database query stored on the MySQL server; it is not a `SELECT` statement but the result set retrieved by that statement. Once a cursor is stored, applications can use it to scroll or browse up and down through the data as needed.

Cursors are used primarily by interactive applications in which users need to scroll up and down through screens of data to browse or make changes.

Note
Only in Stored Procedures Unlike most DBMSs, MySQL cursors may only be used within stored procedures (and functions).

Working with Cursors

Using cursors involves several distinct steps:

1. Before a cursor can be used, it must be declared (defined). This process does not actually involve retrieving any data; it merely involves defining the `SELECT` statement to be used.
2. After it is declared, the cursor must be opened for use. This process involves actually retrieving the data using the previously defined `SELECT` statement.
3. With the cursor populated with data, individual rows can be fetched (retrieved) as needed.
4. When it is done, the cursor must be closed.

After a cursor is declared, it can be opened and closed as often as needed. After it is opened, fetch operations can be performed as often as needed.

Creating Cursors

Cursors are created using the `DECLARE` statement (which you saw in Chapter 23, "Working with Stored Procedures"). `DECLARE` names the cursor and takes a `SELECT` statement, complete with `WHERE` and other clauses, if needed. For example, this statement defines a cursor named `ordernumbers` using a `SELECT` statement that retrieves all orders:

► Input

```
DELIMITER //
CREATE PROCEDURE processorders()
BEGIN
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;
END //
DELIMITER ;
```

► Analysis

This stored procedure does not do a whole lot. A `DECLARE` statement is used to define and name the cursor—in this case `ordernumbers`. Nothing is done with the cursor, and as soon as the stored procedure finishes processing, it will cease to exist (as it is local to the stored procedure).

Note

`DROP` and `CREATE` As you saw in Chapter 23, to update a stored procedure, you need to use `DROP` on it and then use `CREATE`.

Now that the cursor is defined, it is ready to be opened.

Opening and Closing Cursors

Cursors are opened using the `OPEN CURSOR` statement, like this:

Input

```
OPEN ordernumbers;
```

Analysis

When the `OPEN` statement is processed, the query is executed, and the retrieved data is stored for subsequent browsing and scrolling.

After cursor processing is complete, the cursor should be closed using the `CLOSE` statement, as follows:

Input

```
CLOSE ordernumbers;
```

Analysis

`CLOSE` frees up any internal memory and resources used by the cursor, and so every cursor should be closed when it is no longer needed.

After a cursor is closed, it cannot be reused without being opened again. However, a cursor does not need to be declared again to be used; an `OPEN` statement is sufficient.

Note

Implicit Closing If you do not explicitly close a cursor, MySQL will close it automatically when the `END` statement is reached.

Here is an updated version of the previous example:

Input

```
DELIMITER //
CREATE PROCEDURE processorders()
BEGIN
   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;

   -- Open the cursor
   OPEN ordernumbers;

   -- Close the cursor
   CLOSE ordernumbers;

END//

DELIMITER ;
```

► Analysis

This stored procedure declares, opens, and closes a cursor. However, nothing is done with the retrieved data.

Using Cursor Data

After a cursor is opened, each row can be accessed individually using a `FETCH` statement. `FETCH` specifies what is to be retrieved (the desired columns) and where retrieved data should be stored. It also advances the internal row pointer within the cursor so the next `FETCH` statement will retrieve the next row (and not the same one over and over).

The first example retrieves a single row from the cursor (the first row):

► Input

```
DELIMITER //

CREATE PROCEDURE processorders()
BEGIN

   -- Declare local variables
   DECLARE o INT;

   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;

   -- Open the cursor
   OPEN ordernumbers;

   -- Get order number
   FETCH ordernumbers INTO o;

   -- Close the cursor
   CLOSE ordernumbers;

END //

DELIMITER ;
```

► Analysis

Here `FETCH` is used to retrieve the `order_num` column of the current row (starting at the first row automatically) into a local declared variable named `o`. Nothing is done with the retrieved data.

In the next example, the retrieved data is looped through from the first row to the last:

► Input

```
DELIMITER //

CREATE PROCEDURE processorders()
```

```
BEGIN

   -- Declare local variables
   DECLARE done BOOLEAN DEFAULT 0;
   DECLARE o INT;

   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;

   -- Declare continue handler
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;

   -- Open the cursor
   OPEN ordernumbers;

   -- Loop through all rows
   REPEAT

      -- Get order number
      FETCH ordernumbers INTO o;

   -- End of loop
   UNTIL done END REPEAT;

   -- Close the cursor
   CLOSE ordernumbers;

END //

DELIMITER ;
```

► Analysis

Like the previous example, this example uses `FETCH` to retrieve the current `order_num` into a declared variable named `o`. Unlike the previous example, the `FETCH` here is within a `REPEAT`, so it is repeated over and over until `done` is true (as specified by `UNTIL done END REPEAT;`). To make this work, the variable `done` is defined with `DEFAULT 0` (false, not done). So how does `done` get set to true when done? The answer is this statement:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;
```

This statement defines a `CONTINUE HANDLER`, which is code that will be executed when a condition occurs. Here it specifies that when `SQLSTATE '02000'` occurs, `SET done=1`. And `SQLSTATE '02000'` is a *not found* condition, and so it occurs when `REPEAT` cannot continue because there are no more rows to loop through.

Caution

DECLARE Statement Sequence There is a specific order in which DECLARE statements, if used, must be issued. Local variables defined with DECLARE must be defined before any cursors or handlers are defined, and handlers must be defined after any cursors. Failure to follow this sequencing will generate an error message.

If you were to call this stored procedure, it would define variables and a CONTINUE HANDLER, define and open a cursor, repeat through all rows, and then close the cursor.

With this functionality in place, you can now place any needed processing inside the loop—after the FETCH statement and before the end of the loop.

Note

REPEAT or LOOP? In addition to the REPEAT statement used here, MySQL also supports a LOOP statement that can be used to repeat code until LOOP is manually exited using a LEAVE statement. In general, the syntax of the REPEAT statement makes it better suited for looping through cursors.

To put this all together, here is one further revision of our stored procedure with cursor, this time with some actual processing of fetched data:

Input

```
DELIMITER //

CREATE PROCEDURE processorders()
BEGIN
   -- Declare local variables
   DECLARE done BOOLEAN DEFAULT 0;
   DECLARE o INT;
   DECLARE t DECIMAL(8,2);

   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;

   -- Declare continue handler
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;

   -- Create a table to store the results
   CREATE TABLE IF NOT EXISTS ordertotals
      (order_num INT, total DECIMAL(8,2));

   -- Open the cursor
   OPEN ordernumbers;
```

```
    -- Loop through all rows
    REPEAT

       -- Get order number
       FETCH ordernumbers INTO o;

       -- Get the total for this order
       CALL ordertotal(o, 1, t);

       -- Insert order and total into ordertotals
       INSERT INTO ordertotals(order_num, total)
       VALUES(o, t);

    -- End of loop
    UNTIL done END REPEAT;

    -- Close the cursor
    CLOSE ordernumbers;

END //

DELIMITER ;
```

► Analysis

In this example, we've added another variable named `t` (which will store the total for each order). The stored procedure also creates a new table named `ordertotals` on-the-fly (if it does not already exist). This table will store the results generated by the stored procedure. `FETCH` fetches each `order_num` as it did before, and then `CALL` executes another stored procedure (the one created in the previous chapter) to calculate the total with tax for each order (the result of which is stored in `t`). Finally, `INSERT` is used to save the order number and total for each order.

Now the stored procedure can be executed using `CALL`, like this:

► Input

```
CALL processorders();
```

► Analysis

Calling the stored procedure returns no data, but it does create and populate another table that can then be viewed using a simple `SELECT` statement:

► Input

```
SELECT *
FROM ordertotals;
```

► Output

```
+-----------+---------+
| order_num | total   |
+-----------+---------+
```

```
|      20005 |  158.86 |
|      20006 |   58.30 |
|      20007 | 1060.00 |
|      20008 |  132.50 |
|      20009 |   40.78 |
+------------+---------+
```

And there you have it: a complete working example of stored procedures, cursors, row-by-row processing, and even stored procedures calling other stored procedures.

Summary

In this chapter, you learned what cursors are and why they are used. You also saw examples demonstrating basic cursor use, as well as techniques for looping through cursor results and for row-by-row processing.

25

Using Triggers

In this chapter, you'll learn what triggers are and why and how they are used. You'll also learn the syntax for creating and using them.

Understanding Triggers

> **Note**
> **This Chapter Requires MySQL 5** Support For Triggers Was Added To MySQL 5. Therefore, This Chapter Is Applicable To MySQL 5 Or Later Only.

MySQL statements are executed when needed, as are stored procedures. But what if you want a statement (or statements) to be executed automatically when events occur? For example:

- Every time a customer is added to a database table, you want to check that the phone number is formatted correctly and that the state abbreviation is in uppercase.
- Every time a product is ordered, you want to subtract the ordered quantity from the number in stock.
- Whenever a row is deleted, you want to save a copy in an archive table.

What all these examples have in common is that they need to be processed automatically whenever a table change occurs. And that is exactly what triggers do. A *trigger* is a MySQL statement (or a group of statements enclosed within `BEGIN` and `END` statements) that is automatically executed by MySQL in response to any of these statements:

- `DELETE`
- `INSERT`
- `UPDATE`

No other MySQL statements support triggers.

Creating Triggers

When creating a trigger, you need to specify four pieces of information:

- The unique trigger name
- The table with which the trigger is to be associated
- The action that the trigger should respond to (`DELETE`, `INSERT`, or `UPDATE`)
- When the trigger should be executed (before or after processing)

Tip

Keep Trigger Names Unique per Database Unlike most DBMSs, MySQL requires trigger names to be unique per table but not per database. This means that two tables in the same database can have triggers of the same name.

Triggers are created using the `CREATE TRIGGER` statement. Here is a really simple (and not overly useful) example:

Input

```
CREATE TRIGGER newproduct AFTER INSERT ON products
FOR EACH ROW SET @result = 1;
```

Analysis

Here `CREATE TRIGGER` is used to create the new trigger named `newproduct`.

Triggers can be executed before or after an operation occurs, and here `AFTER INSERT` is specified so the trigger will execute after a successful `INSERT` statement has been executed. The trigger then specifies `FOR EACH ROW` and the code to be executed for each inserted row. In this example, a variable named `@result` will be set to `1`.

To test this trigger, use the `INSERT` statement to add one or more rows to `products`. You can then use `SELECT` to display the variable.

Note

Only Tables Triggers are only supported on tables, not on views. (They are also not supported on temporary tables.)

Triggers are defined per time per event per table, and only one trigger per time per event per table is allowed. Up to six triggers are supported per table (before and after `INSERT`, `UPDATE`, and `DELETE`). A single trigger cannot be associated with multiple events or multiple tables, so if you need triggers to be executed for both `INSERT` and `UPDATE` operations, you'll need to define two triggers.

Note
When Triggers Fail When a `BEFORE` trigger fails, MySQL does not perform the requested operation. In addition, when either a `BEFORE` trigger or the statement itself fails, MySQL does not execute an `AFTER` trigger (if one exists).

Dropping Triggers

By now the syntax for dropping a trigger should be apparent. To drop a trigger, you use the `DROP TRIGGER` statement, as shown here:

► Input

```
DROP TRIGGER newproduct;
```

► Analysis

Triggers cannot be updated or overwritten. To modify a trigger, you must drop it and re-create it.

Using Triggers

Now that we've covered the basics of triggers, we will now look at each of the supported trigger types and the differences between them.

INSERT Triggers

An `INSERT` trigger is executed before or after an `INSERT` statement is executed. Be aware of the following:

- In `INSERT` trigger code, you can refer to a virtual table named `NEW` to access the rows being inserted.
- In a `BEFORE INSERT` trigger, the values in `NEW` may also be updated (so you can change values that are about to be inserted).
- For `AUTO_INCREMENT` columns, `NEW` contains `0` before the trigger and the new automatically generated value after the trigger.

Let's look at an example—a really useful one. `vendors` contains a column named `vend_state` that stores a vendor's state (as part of its address). Ideally state abbreviations should be capitalized, such as `CA` rather than `ca` or `Ca`. You could ask users to always enter the data correctly, but, yeah right! Using a trigger is a better solution:

► Input

```
DELIMITER //

CREATE TRIGGER newvendor AFTER INSERT ON vendors
FOR EACH ROW
```

```
BEGIN
   UPDATE vendors SET vend_state=Upper(vend_state) WHERE vend_id = NEW.vend_id;
END //

DELIMITER ;
```

Analysis

This code creates a trigger named `newvendor` that is executed by `AFTER INSERT ON vendors`. When a new vendor is saved in `vendors`, a copy is saved in `NEW`, and `NEW.vend_id` contains the ID of the newly inserted vendor. The trigger contains an `UPDATE` statement that updates `vend_state` with `Upper(vend_state)` and uses `NEW.vend_id` in the `WHERE` clause so that the right row is updated. Now, no matter what the user enters, the data will be stored correctly.

To test this trigger, add a new vendor and then use `SELECT` to verify that `vend_state` was indeed updated.

Tip

BEFORE or AFTER? As a rule, you should use `BEFORE` for any data validation and cleanup to ensure that the data inserted into the table is exactly as needed. This applies to `UPDATE` triggers, too.

DELETE Triggers

A `DELETE` trigger is executed before or after a `DELETE` statement is executed. Be aware of the following:

- Within `DELETE` trigger code, you can refer to a virtual table named `OLD` to access the rows being deleted.
- The values in `OLD` are all read-only and cannot be updated.

The following example demonstrates the use of `OLD` to save rows that are about to be deleted into an archive table:

Input

```
DELIMITER //

CREATE TRIGGER deleteorder BEFORE DELETE ON orders
FOR EACH ROW
BEGIN
   INSERT INTO archive_orders(order_num,
                              order_date,
                              cust_id)
   VALUES(OLD.order_num,
          OLD.order_date,
          OLD.cust_id);
```

```
END//

DELIMITER ;
```

► Analysis

Before any order is deleted, this trigger is executed. It uses an `INSERT` statement to save the values in `OLD` (the order about to be deleted) into an archive table named `archive_orders`. (To actually use this example, you need to create a table named `archive_orders` with the same columns as `orders`.)

The advantage of using a `BEFORE DELETE` trigger (as opposed to an `AFTER DELETE` trigger) is that if, for some reason, the order cannot be archived, the `DELETE` will be aborted.

> **Note**
> **Multi-Statement Triggers** Notice that the trigger `deleteorder` uses `BEGIN` and `END` statements to mark the trigger body. This is actually not necessary in this example, although it does no harm being there. The advantage of using a `BEGIN END` block is that the trigger can then accommodate multiple SQL statements (one after the other within the `BEGIN END` block).

UPDATE Triggers

An `UPDATE` trigger is executed before or after an `UPDATE` statement is executed. Be aware of the following:

- Within `UPDATE` trigger code, you can refer to a virtual table named `OLD` to access the previous (pre-`UPDATE` statement) values and `NEW` to access the new updated values.
- In a `BEFORE UPDATE` trigger, the values in `NEW` may also be updated so that you can change the values that are about to be used in the `UPDATE` statement.
- The values in `OLD` are all read-only and cannot be updated.

The following example revisits the `INSERT` example used previously and ensures that state abbreviations are always in uppercase, even when updated:

► Input

```
DELIMITER //

CREATE TRIGGER updatevendor BEFORE UPDATE ON vendors
FOR EACH ROW
SET NEW.vend_state = Upper(NEW.vend_state) //

DELIMITER ;
```

► Analysis

This version works a little differently than the earlier one. Rather than save the row and then update it, this version does data cleanup before using `BEFORE UPDATE`. Each time a row is updated, the value in `NEW.vend_state` (the value that will be used to update table rows) is replaced with `Upper(NEW.vend_state)`.

More on Triggers

Before wrapping up this chapter, here are some important points to keep in mind when using triggers:

- Creating triggers might require special security access. However, trigger execution is automatic. If an `INSERT`, `UPDATE`, or `DELETE` statement is executed, any associated triggers are executed, too.
- Triggers should be used to ensure data consistency (case, formatting, and so on). The advantage of performing this type of processing in a trigger is that it always happens, and it happens transparently, regardless of the client application.
- One very interesting use for triggers is to create an audit trail. By using triggers, it would be very easy to log changes (even before and after states, if needed) to another table.
- Unfortunately, the `CALL` statement is not supported in MySQL triggers. This means that stored procedures cannot be invoked from within triggers. Any needed stored procedure code needs to be replicated within the trigger itself.

Summary

In this chapter, you learned what triggers are and why they are used. You learned about the trigger types and when they can be executed. You also saw examples of triggers used for `INSERT`, `DELETE`, and `UPDATE` operations.

26

Managing Transaction Processing

In this chapter, you'll learn what transactions are and how to use `COMMIT` and `ROLLBACK` statements to manage transaction processing.

Understanding Transaction Processing

Note

Not All Engines Support Transactions—As explained in Chapter 21, "Creating and Manipulating Tables," MySQL supports the use of several underlying database engines. Not all engines support explicit management of transaction processing, as will be explained in this chapter. The two most commonly used engines are MyISAM and InnoDB. The former does not support explicit transaction management and the latter does. This is why the sample tables used in this book were created to use InnoDB instead of the more commonly used MyISAM. If you need transaction processing functionality in your applications, be sure to use the correct engine type.

Transaction processing is used to maintain database integrity by ensuring that batches of MySQL operations execute completely or not at all.

As explained back in Chapter 15, "Joining Tables," relational databases are designed so data is stored in multiple tables to facilitate easier data manipulation, management, and reuse. Without going into the hows and whys of relational database design, take it as a given that well-designed database schemas are relational to some degree.

The orders tables you've been using in prior chapters provide a good example of this. Orders are stored in two tables: `orders` stores orders, and `orderitems` stores the individual items ordered. These two tables are related to each other using unique IDs called *primary keys* (as discussed in Chapter 1, "Understanding SQL"). These tables, in turn, are related to other tables containing customer and product information.

The process of adding an order to the system is as follows:

1. Check if the customer is already in the database (that is, present in the `customers` table). If not, add them.
2. Retrieve the customer's ID.
3. Add a row to the `orders` table that associates it with the customer ID.
4. Retrieve the new order ID assigned in the `orders` table.
5. Add one row to the `orderitems` table for each item ordered and associate it with the `orders` table by the retrieved ID (and with the `products` table by the product ID).

Now imagine that some database failure (for example, lack of disk space, security restrictions, table locks) prevents this entire sequence from completing. What happens to your data?

Well, if the failure occurs after the customer is added and before the `orders` table is added, there is no real problem. It is perfectly valid to have customers without orders. When you run the sequence again, the inserted customer record will be retrieved and used. You can effectively pick up where you left off.

But what if the failure occurs after the `orders` row is added and before the `orderitems` rows are added? Now you have an empty order sitting in your database.

Worse, what if the system fails during the process of adding the `orderitems` rows? Now you end up with a partial order in your database, but you don't know it.

How do you solve this problem? This is where transaction processing comes in. *Transaction processing* is a mechanism used to manage sets of MySQL operations that must be executed in batches to ensure that databases never contain the results of partial operations. With transaction processing, you can ensure that sets of operations are not aborted mid-processing; they either execute in their entirety or not at all (unless explicitly instructed otherwise). If no error occurs, the entire set of statements is committed (written) to the database tables. If an error does occur, a rollback (undo) can occur to restore the database to a known and safe state.

So, in the context of this example, here is how the process works:

1. Check if the customer is already in the database. If not, add them.
2. Commit the customer information.
3. Retrieve the customer's ID.
4. Add a row to the `orders` table.
5. If a failure occurs while adding the row to `orders`, roll back.
6. Retrieve the new order ID assigned in the `orders` table.
7. Add one row to the `orderitems` table for each item ordered.
8. If a failure occurs while adding rows to `orderitems`, roll back all the `orderitems` rows added and the `orders` row.
9. Commit the order information.

When working with transactions and transaction processing, you need to know a few important terms:

- **Transaction:** A block of SQL statements
- **Rollback:** The process of undoing specified SQL statements
- **Commit:** The process of writing unsaved SQL statements to the database tables
- **Savepoint:** A temporary placeholder in a transaction set to which you can issue a rollback (as opposed to rolling back an entire transaction)

Controlling Transactions

Now that you know what transaction processing is, let's look at what is involved in managing transactions.

The key to managing transactions involves breaking your SQL statements into logical chunks and explicitly stating when data should be rolled back and when it should not.

The MySQL statement used to mark the start of a transaction is:

► Input

```
START TRANSACTION;
```

Using ROLLBACK

The MySQL `ROLLBACK` command is used to roll back (undo) MySQL statements, as shown in this statement:

► Input

```
SELECT * FROM ordertotals;
START TRANSACTION;
DELETE FROM ordertotals;
SELECT * FROM ordertotals;
ROLLBACK;
SELECT * FROM ordertotals;
```

► Analysis

Granted, this is not a particularly useful example, but it does help demonstrate the logical flow of a SQL transaction. This example starts by displaying the contents of the `ordertotals` table (which was populated in Chapter 24, "Using Cursors"). First, a `SELECT` is performed to show that the table is not empty. Then a transaction is started, and all of the rows in `ordertables` are deleted with a `DELETE` statement. Another `SELECT` verifies that, indeed, `ordertotals` is empty. Then a `ROLLBACK` statement is used to roll back all statements until `START TRANSACTION`, and the final `SELECT` shows that the table is no longer empty.

Obviously, `ROLLBACK` can only be used within a transaction (after a `START TRANSACTION` command has been issued).

Tip

Which Statements Can You Roll Back? Transaction processing is used to manage `INSERT`, `UPDATE`, and `DELETE` statements. You cannot roll back `SELECT` statements. (There would not be much point in doing so anyway.) You cannot roll back `CREATE` or `DROP` operations. These statements may be used in a transaction block, but if you perform a rollback, they will not be undone.

Using COMMIT

MySQL statements are usually executed and written directly to the database tables. A commit (write or save) operation that happens automatically is known as an *implicit commit*.

Within a transaction block, however, commits do not occur implicitly. To force an explicit commit, the `COMMIT` statement is used, as shown here:

► Input

```
START TRANSACTION;
DELETE FROM orderitems WHERE order_num = 20010;
DELETE FROM orders WHERE order_num = 20010;
COMMIT;
```

► Analysis

In this example, order number 20010 is deleted from the system entirely. Because this involves updating two database tables, `orders` and `orderitems`, a transaction block is used to ensure that the order is not partially deleted; you wouldn't want data deleted from one table but not the other. The final `COMMIT` statement writes the change only if no error occurred. If the first `DELETE` worked but the second failed, the `DELETE` would not be committed; rather, it would effectively be automatically undone.

Note

Implicit Transaction Closes After a `COMMIT` or `ROLLBACK` statement has been executed, the transaction is automatically closed (and future changes are implicitly committed).

Using Savepoints

Simple `ROLLBACK` and `COMMIT` statements enable you to write or undo an entire transaction. Although this works for simple transactions, more complex transactions might require partial commits or rollbacks.

For example, the process of adding an order described previously is a single transaction. If an error occurs, you only want to roll back to the point before the `orders` row was added. You do not want to roll back the addition to the `customers` table (if there was one).

To support the rollback of partial transactions, you must be able to put placeholders at strategic locations in the transaction block. Then, if a rollback is required, you can roll back to one of the placeholders.

These placeholders are called *savepoints*, and to create one, you use the `SAVEPOINT` statement, as follows:

Input

```
SAVEPOINT delete1;
```

Each savepoint has a unique name that identifies it so that, when you roll back, MySQL knows where you are rolling back to. To roll back to the savepoint just created, use the following:

Input

```
ROLLBACK TO delete1;
```

Tip

The More Savepoints, the Better You can have as many savepoints as you like within your MySQL code, and the more, the better. Why? Because the more savepoints you have, the more flexibility you have in managing rollbacks exactly as you need to.

Note

Releasing Savepoints A savepoint is automatically released after a transaction completes (that is, after a `ROLLBACK` or `COMMIT` is issued). As of MySQL 5, you can also explicitly release a savepoint by using `RELEASE SAVEPOINT`.

Changing the Default Commit Behavior

As already explained, the default MySQL behavior is to automatically commit any and all changes. In other words, any time you execute a MySQL statement, that statement is actually being performed against the tables, and the changes are made immediately. To instruct MySQL to not automatically commit changes, you need to use the following statement:

Input

```
SET autocommit=0;
```

Analysis

The `autocommit` flag determines whether changes are committed automatically without requiring a manual `COMMIT` statement. Setting `autocommit` to `0` (false) instructs MySQL to not automatically commit changes (until the flag is set back to `1`, or true).

Note

Flag Is Connection Specific The `autocommit` flag is per connection, not server-wide.

Summary

In this chapter, you learned that transactions are blocks of SQL statements that must be executed as a batch. You learned how to use the `COMMIT` and `ROLLBACK` statements to explicitly manage when data is written and when it is undone. You also learned how to use savepoints to provide a greater level of control over rollback operations.

27

Globalization and Localization

In this chapter, you'll learn the basics of how MySQL handles different character sets and languages.

Understanding Character Sets and Collation Sequences

Database tables are used to store and retrieve data. Different languages and character sets need to be stored and retrieved differently. Therefore, MySQL needs to accommodate different character sets (different alphabets and characters) as well as different ways to sort and retrieve data.

When discussing multiple languages and characters sets, you will run into the following important terms:

- **Character set:** A collection of letters and symbols
- **Encoding:** The internal representation of the members of a character set
- **Collation:** Instructions that dictate how characters are to be compared

Note

Why Collations Are Important Sorting text in English is easy, right? Well, maybe not. Consider the words *APE*, *apex*, and *Apple*. Are they in the correct sorted order? It depends on whether you want a case-sensitive sorting or a sorting that is not case-sensitive. The words would be sorted one way using a case-sensitive collation and another way for a collation that isn't case-sensitive. And this affects more than just sorting (as in data sorted using `ORDER BY`); it also affects searches (whether or not a `WHERE` clause looking for *apple* finds *APPLE*, for example). The situation gets even more complex when characters such as the French à or German ö are used, and it becomes even more complex when non-Latin-based character sets are used (Japanese, Hebrew, Russian, and so on).

In MySQL there is not much to worry about during regular database activity (`SELECT`, `INSERT`, and so forth). Rather, the decisions about which character set and collation to use occur at the server, database, and table level.

Working with Character Sets and Collation Sequences

MySQL supports a vast number of character sets. To see the complete list of supported character sets, use this statement:

► Input

```
SHOW CHARACTER SET;
```

► Analysis

This statement displays all available character sets, along with a description of each one and the default collation for each one.

To see the complete list of supported collations, use this statement:

► Input

```
SHOW COLLATION;
```

► Analysis

This statement displays all available collations, along with the character sets to which each one applies. Several character sets have more than one collation. `latin1`, for example, has several character sets for different European languages, and many sets appear twice: They appear once case-sensitive (designated by `_cs`) and once not case-sensitive (designated by `_ci`).

A default character set and collation are defined (usually by the system administration at installation time). In addition, when databases are created, default character sets and collations may be specified, too. To determine the character sets and collations in use, use these statements:

► Input

```
SHOW VARIABLES LIKE 'character%';
SHOW VARIABLES LIKE 'collation%';
```

In practice, character sets can seldom be server-wide (or even database-wide) settings. Different tables, and even different columns, may require different character sets, and so both may be specified when a table is created.

To specify a character set and collation for a table, you use `CREATE TABLE` (seen in Chapter 21, "Creating and Manipulating Tables") with additional clauses:

► Input

```
CREATE TABLE mytable
(
```

```
   column1   INT,
   column2   VARCHAR(10)
) DEFAULT CHARACTER SET hebrew
  COLLATE hebrew_general_ci;
```

► Analysis

This statement creates a two-column table and specifies both a character set and a collate sequence.

In this example, both `CHARACTER SET` and `COLLATE` are specified, but if only one of them (or neither of them) is specified, this is how MySQL determines what to use:

- If both `CHARACTER SET` and `COLLATE` are specified, those values are used.
- If only `CHARACTER SET` is specified, it is used along with the default collation for that character set (as specified in the `SHOW CHARACTER SET` results).
- If neither `CHARACTER SET` nor `COLLATE` is specified, the database default is used.

In addition to being able to specify character set and collation table-wide, MySQL also allows them to be set per column, as seen here:

► Input

```
CREATE TABLE mytable
(
   column1   INT,
   column2   VARCHAR(10),
   column3    VARCHAR(10) CHARACTER SET latin1
                          COLLATE latin1_general_ci
) DEFAULT CHARACTER SET hebrew
  COLLATE hebrew_general_ci;
```

► Analysis

Here `CHARACTER SET` and `COLLATE` are specified for the entire table as well as for a specific column.

As mentioned previously, the collation plays a key role in sorting data that is retrieved with an `ORDER BY` clause. If you need to sort specific `SELECT` statements by using a collation sequence other than the one used at table creation time, you may do so in the `SELECT` statement:

► Input

```
SELECT * FROM customers
ORDER BY lastname, firstname
   COLLATE latin1_general_cs;
```

► Analysis

This `SELECT` uses `COLLATE` to specify an alternate collation sequence (in this example, a case-sensitive one). This will obviously affect the order in which results are sorted.

Tip

Occasional Case-Sensitivity The `SELECT` statement just shown demonstrates a useful technique for performing case-sensitive searches on a table that is usually not case-sensitive. And, of course, the reverse works just as well.

Note

Other `SELECT COLLATE` Clauses In addition to being used in `ORDER BY` clauses, as shown here, `COLLATE` can be used with `GROUP BY`, `HAVING`, aggregate functions, aliases, and more.

One final point worth noting is that strings may be converted between character sets if absolutely needed. To do this, use the `Cast()` or `Convert()` functions.

Summary

In this chapter, you learned the basics of character sets and collations. You also learned how to define the character sets and collations for specific tables and columns and how to use alternate collations when needed.

28

Managing Security

In this chapter, you'll learn about MySQL access control and user management. Database servers usually contain critical data, and ensuring the safety and integrity of that data requires that access control be used.

Understanding Access Control

The basis of security for your MySQL server is this: *Users should have appropriate access to the data they need—no more and no less*. In other words, users should not have too much access to too much data.

Consider the following:

- Most users need to read and write data from tables, but few users will ever need to be able to create and drop tables.
- Some users might need to read tables but might not need to update them.
- You might want to allow users to add data but not delete data.
- Some users (managers or administrators) might need rights to manipulate user accounts, but most do not.
- You might want users to access data via stored procedures but never directly.
- You might want to restrict access to some functionality based on the user's login location.

These are just examples, but they help demonstrate an important point: You need to provide users with the access they need—and just the access they need. This is known as *access control*, and managing access control requires creating and managing user accounts.

Tip

Use MySQL Administrator MySQL Workbench (described in Chapter 2, "Introducing MySQL") provides a graphical user interface that can be used to manage users and account rights. Internally, MySQL Workbench uses the statements described in this chapter, enabling you to manage access control interactively and simply.

Back in Chapter 3, "Working with MySQL," you learned that you need to log in to MySQL in order to perform any operations. When it is first installed, MySQL creates a user account named `root` that has complete and total control over the entire MySQL server. You might have been using the `root` login throughout the chapters in this book, and that is fine when experimenting with MySQL on non-live servers. But in the real world, you'd never use `root` on a day-to-day basis. Instead, you'd create a series of accounts for administration, for users, for developers, and so on.

Note

Preventing Innocent Mistakes It is important to note that access control is not just intended to keep out users with malicious intent. More often than not, data nightmares are the result of inadvertent mistakes, mistyped MySQL statements, being in the wrong database, or other user errors. Access control helps avoid these situations by ensuring that users are unable to execute statements they should not be executing.

Caution

Don't Use `root` The `root` login should be considered sacred. Use it only when absolutely needed (perhaps when you cannot get into other administrative accounts). `root` should never be used in day-to-day MySQL operations.

Managing Users

MySQL user accounts and information are stored in a MySQL database named `mysql`. You usually do not need to access the `mysql` database and tables directly (as you will soon see), but sometimes you might. One of those times is when you want to obtain a list of all user accounts. To do that, use the following code:

► Input

```
USE mysql;
SELECT user FROM user;
```

► Output

```
+------+
| user |
+------+
| root |
+------+
```

► Analysis

The `mysql` database contains a table named `user` that contains all user accounts. `user` contains a column named `user` that contains the user login name. A newly installed server might have a single user listed (as shown in this example) or several default accounts; established servers are likely to have far more users.

Tip

Test Using Multiple Clients The easiest way to test changes made to user accounts and rights is to open multiple database clients (multiple copies of the `mysql` command-line utility, for example), one logged in with the administrative login and the others logged in as the users being tested.

Creating User Accounts

To create a new user account, use the `CREATE USER` statement, as shown here:

Input

```
CREATE USER ben IDENTIFIED BY 'p@$$w0rd';
```

Analysis

`CREATE USER` creates a new user account. A password need not be specified at user account creation time, but this example does specify a password, using `IDENTIFIED BY 'p@$$w0rd'`.

If you were to list the user accounts again, you'd see the new account listed in the output.

Tip

Specifying a Hashed Password The password specified by `IDENTIFIED BY` is plain-text that MySQL will encrypt before saving the password in the user table. To specify the password as a hashed value, use `IDENTIFIED BY PASSWORD` instead.

Note

Using `GRANT` or `INSERT` The `GRANT` statement (which we will get to shortly) can also create user accounts, but generally `CREATE USER` is the cleanest and simplest option. In addition, it is possible to add users by inserting rows into `user` directly, but to be safe, doing this is generally not recommended. The tables that MySQL uses to store user account information (as well as table schemas and more) are extremely important, and any damage to them could seriously harm the MySQL server. Therefore, it is always better to use tags and functions to manipulate these tables than to manipulate them directly.

To rename a user account, use the `RENAME USER` statement, like this:

Input

```
RENAME USER ben TO bforta;
```

> **Note**
> **Before MySQL 5** `RENAME USER` is supported only in MySQL 5 or later. To rename a user in earlier versions of MySQL, use `UPDATE` to update the `user` table directly.

Deleting User Accounts

To delete a user account (along with any associated rights and privileges), use the `DROP USER` statement, as shown here:

► Input

```
DROP USER bforta;
```

> **Note**
> **Before MySQL 5** As of MySQL 5, `DROP USER` deletes user accounts and all associated account rights and privileges. Prior to MySQL 5, `DROP USER` could only be used to drop user accounts with no associated account rights and privileges. Therefore, if you are using an older version of MySQL, you will need to first remove associated account rights and privileges by using `REVOKE` and then use `DROP USER` to delete the account.

Setting Access Rights

Once user accounts are created, you need to assign access rights and privileges. Newly created user accounts have no access at all. They can log in to MySQL but will see no data and will be unable to perform any database operations.

To see the rights granted to a user account, use `SHOW GRANTS FOR`, as shown in this example:

► Input

```
SHOW GRANTS FOR bforta;
```

► Output

```
+---------------------------------------------------+
| Grants for bforta@%                               |
+---------------------------------------------------+
| GRANT USAGE ON *.* TO 'bforta'@'%'                |
+---------------------------------------------------+
```

► Analysis

The output shows that user `bforta` has a single right granted, `USAGE ON *.*`. `USAGE` means no *rights at all* (not overly intuitive, I know), so the results mean *no rights to anything on any database and any table*.

> **Note**
> **Users Are Defined as *user@host*** MySQL privileges are defined using a combination of username and hostname. If no hostname is specified, the default hostname `%` is used (effectively granting access to the user regardless of the hostname).

To set rights, you use the `GRANT` statement. At a minimum, `GRANT` requires that you specify the following:

- The privilege being granted
- The database or table being granted access to
- The username

The following example demonstrates the use of `GRANT`:

► Input

```
GRANT SELECT ON crashcourse.* TO bforta;
```

► Analysis

This statement allows the use of `SELECT` on `crashcourse.*` (that is, the `crashcourse` database, including all its tables). By granting `SELECT` access only, user `bforta` has read-only access to all data in the `crashcourse` database.

By using `SHOW GRANTS`, you can see that this change was made:

► Input

```
SHOW GRANTS FOR bforta;
```

► Output

```
+-------------------------------------------------+
| Grants for bforta@%                             |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'bforta'@'%'              |
| GRANT SELECT ON 'crashcourse'.* TO 'bforta'@'%' |
+-------------------------------------------------+
```

► Analysis

Each `GRANT` adds (or updates) a permission statement for the user. MySQL reads all of the grants and determines the rights and permissions based on them.

The opposite of `GRANT` is `REVOKE`, which is used to revoke specific rights and permissions. Here is an example:

► Input

```
REVOKE SELECT ON crashcourse.* FROM bforta;
```

► Analysis

This `REVOKE` statement takes away the `SELECT` access just granted to user `bforta`. The access being revoked must exist, or an error will be thrown.

GRANT and REVOKE can be used to control access at several levels:

- Entire server, using GRANT ALL and REVOKE ALL
- Entire database, using ON database.*
- Specific tables, using ON database.table
- Specific columns
- Specific stored procedures

Table 28.1 lists the rights and privileges that may be granted or revoked.

TABLE 28.1 Rights and Privileges

Privilege	Description
ALL	All privileges except GRANT OPTION
ALTER	Use of ALTER TABLE
ALTER ROUTINE	Use of ALTER PROCEDURE and DROP PROCEDURE
CREATE	Use of CREATE TABLE
CREATE ROUTINE	Use of CREATE PROCEDURE
CREATE TEMPORARY TABLES	Use of CREATE TEMPORARY TABLE
CREATE USER	Use of CREATE USER, DROP USER, RENAME USER, and REVOKE ALL PRIVILEGES
CREATE VIEW	Use of CREATE VIEW
DELETE	Use of DELETE
DROP	Use of DROP TABLE
EXECUTE	Use of CALL and stored procedures
FILE	Use of SELECT INTO OUTFILE and LOAD DATA INFILE
GRANT OPTION	Use of GRANT and REVOKE
INDEX	Use of CREATE INDEX and DROP INDEX
INSERT	Use of INSERT
LOCK TABLES	Use of LOCK TABLES
PROCESS	Use of SHOW FULL PROCESSLIST
RELOAD	Use of FLUSH
REPLICATION CLIENT	Access to location of servers
REPLICATION SLAVE	Use by replication slaves
SELECT	Use of SELECT
SHOW DATABASES	Use of SHOW DATABASES
SHOW VIEW	Use of SHOW CREATE VIEW

Privilege	Description
SHUTDOWN	Use of `mysqladmin shutdown` (used to shut down MySQL)
SUPER	Use of `CHANGE MASTER`, `KILL`, `LOGS`, `PURGE MASTER`, and `SET GLOBAL`. Also allows `mysqladmin debug` login.
UPDATE	Use of `UPDATE`
USAGE	No access

By using `GRANT` and `REVOKE` in conjunction with the privileges listed in Table 28.1, you have complete control over what users can and cannot do with your precious data.

Note

Granting for the Future When using `GRANT` and `REVOKE`, the user account must exist, but the objects being referred to need not. This allows administrators to design and implement security before databases and tables are even created.

A side effect of this is that if a database or table is removed (with a `DROP` statement), any associated access rights still exist. And if the database or table is re-created in the future, those rights will again apply to the database or table.

Tip

Simplifying Multiple Grants You can string together multiple `GRANT` statements by listing the privileges and comma delimiting them, as shown in this example:

```
GRANT SELECT, INSERT ON crashcourse.* TO bforta;
```

Changing Passwords

To change user passwords, you use the `SET PASSWORD` statement. A new password must be encrypted by being passed to the `Password()` function, as shown here:

Input

```
SET PASSWORD FOR bforta = Password('n3w p@$$w0rd');
```

Analysis

`SET PASSWORD` updates the user password.

You can also use `SET PASSWORD` to set your own password:

Input

```
SET PASSWORD = Password('n3w p@$$w0rd');
```

► Analysis

When no username is specified, `SET PASSWORD` updates the password for the user who is currently logged in.

Summary

In this chapter, you learned about access control and how to secure a MySQL server by assigning specific rights.

29

Database Maintenance

In this chapter, you'll learn how to perform common database maintenance tasks.

Backing Up Data

Like all other data, MySQL data must be backed up regularly. Because MySQL databases are disk-based files, normal backup systems and routines can back up MySQL data. However, those files are always open and in use, so normal file copy backups might not always work.

Here are possible solutions to this problem:

- Use the command-line utility `mysqldump` to dump all database contents to an external file. This utility should ideally be run before regular backups occur so the dumped file will be backed up properly.
- Use the command-line utility `mysqlhotcopy` to copy all data from a database. (This utility is not supported by all database engines.)
- Use MySQL to dump all data to an external file using `BACKUP TABLE` or `SELECT INTO OUTFILE`. Both statements take the name of a system file to be created; and that file must not already exist, or an error will be generated. Data can be restored by using `RESTORE TABLE`.

Tip

Flush Unwritten Data First To ensure that all data is written to disk (including any index data), you might need to use a `FLUSH TABLES` statement before performing your backup.

Performing Database Maintenance

MySQL features a series of statements that can (and should) be used to ensure that databases are correct and functioning properly.

Here are some statements you should be aware of:

- `ANALYZE TABLE` is used to check that table keys are correct. `ANALYZE TABLE` returns status information, as shown here:

Input

```
ANALYZE TABLE orders;
```

Output

```
+--------------------+---------+----------+----------+
| Table              | Op      | Msg_type | Msg_text |
+--------------------+---------+----------+----------+
| crashcourse.orders | analyze | status   | OK       |
+--------------------+---------+----------+----------+
```

- `CHECK TABLE` is used to check tables for a variety of problems. Indexes are also checked on a MyISAM table. `CHECK TABLE` supports a series of modes for use with MyISAM tables. `CHANGED` checks tables that have changed since the last check, `EXTENDED` performs the most thorough check, `FAST` checks only tables that were not closed properly, `MEDIUM` checks all deleted links and performs key verification, and `QUICK` performs a quick scan only. In this example, `CHECK TABLE` finds and repairs a problem:

Input

```
USE crashcourse;
CHECK TABLE orders, orderitems;
```

Output

```
+-----------------------+-------+----------+-----------------------+
| Table                 | Op    | Msg_type | Msg_text              |
+-----------------------+-------+----------+-----------------------+
| crashcourse.orders    | check | status   | OK                    |
| crashcourse.orderitems | check | warning  | Table is marked as    |
|                       |       |          | crashed               |
| crashcourse.orderitems | check | status   | OK                    |
+-----------------------+-------+----------+-----------------------+
```

- If MyISAM table access produces incorrect and inconsistent results, you might need to repair the table by using `REPAIR TABLE`. This statement should not be used frequently, and if regular use is required, there is likely a far bigger problem that needs to be addressed.
- If you delete large amounts of data from a table, you should use `OPTIMIZE TABLE` to reclaim previously used space and optimize the performance of the table.

Diagnosing Startup Problems

Server startup problems usually occur when a change has been made to the MySQL configuration or to the server. MySQL reports errors when startup problems occur, but because most MySQL servers are started automatically as system processes or services, these messages might not be seen.

When troubleshooting system startup problems, try to manually start the server first. You start the MySQL server by executing `mysqld` on the command line. Here are several important command-line options for `mysqld`:

- `--help` displays help as a list of options.
- `--safe-mode` loads the server minus some optimizations.
- `--verbose` displays full text messages. You can use it in conjunction with `--help` for more detailed help messages.
- `--version` displays version information and then quits.

Several additional command-line options (pertaining to the use of log files) are listed in the next section.

Reviewing Log Files

MySQL maintains a series of log files that administrators rely on extensively. These are the primary log files:

- **Error log:** This log contains details about startup and shutdown problems and any critical errors. It is usually named `hostname.err` and located in the `data` directory. You can change this name by using the `--log-error` command-line option.
- **Query log:** This log details all MySQL activity and can be very useful in diagnosing problems. This log file can get very large very quickly, so it should not be used for extended periods of time. It is usually named `hostname.log` and located in the `data` directory. You can change this name by using the `--log` command-line option.
- **Binary log:** This log logs all statements that updated (or could have updated) data. It is usually named `hostname-bin` and located in the `data` directory. You can change this name by using the `--log-bin` command-line option. Note that this log file was added in MySQL 5; the update log is used in earlier versions of MySQL.
- **Slow query log:** As its name suggests, this log details any queries that execute slowly. This log can be useful in determining where database optimizations are needed. It is usually named `hostname-slow.log` and located in the `data` directory. You can change this name by using the `--log-slow-queries` command-line option.

When logging is being done, you can use the `FLUSH LOGS` statement to flush and restart all log files.

Tip

Use MySQL Workbench for DBMS Maintenance Throughout this book, we've used MySQL Workbench primarily to write and execute SQL statements. But MySQL Workbench has evolved into more than just a SQL editor. In fact, the Server menu and the Administration tab of the Navigator provides interactive access to all of the commands listed above—and more.

Summary

In this chapter, you learned about some basic MySQL database maintenance tools and techniques.

30

Improving Performance

In this chapter, you'll review some important points pertaining to the performance of MySQL.

Improving Performance

Database administrators spend a significant portion of their lives tweaking and experimenting to improve DBMS performance. Poorly performing databases (and database queries, for that matter) tend to be the most frequent culprits when diagnosing application sluggishness and performance problems.

What follows is not, by any stretch of the imagination, the last word on MySQL performance. This chapter is intended to review key points made in the previous 29 chapters, as well as to provide a springboard from which to launch discussion and analysis of performance optimization.

So, here goes:

- First and foremost, MySQL (like all other DBMSs) has specific hardware recommendations. Using any old computer as a database server is fine when learning and playing with MySQL. But production servers should adhere to all recommendations.
- As a rule, a critical production DBMS should run on its own dedicated server.
- MySQL is preconfigured with a series of default settings that are usually a good place to start. But after a while, you might need to tweak memory allocation, buffer sizes, and more. (To see the current settings, use `SHOW VARIABLES;` and `SHOW STATUS;`.)
- MySQL is a multi-user multi-threaded DBMS, which means it often performs multiple tasks at the same time. If one of those tasks is executing slowly, all requests will suffer. If you are experiencing unusually poor performance, use `SHOW PROCESSLIST` to display all active processes (along with their thread IDs and execution times). You can also use the `KILL` command to terminate a specific process. (You need to be logged in as an administrator to use `KILL`.)
- There is almost always more than one way to write a `SELECT` statement. Experiment with joins, unions, subqueries, and more to find what is optimum for you and your data.

- Use the `EXPLAIN` statement to have MySQL explain how it will execute a `SELECT` statement.
- As a general rule, stored procedures execute more quickly than individual MySQL statements.
- Use the right data types—always.
- Never retrieve more data than you need. For example, don't use `SELECT *` unless you truly need each and every column.
- Some operations (including `INSERT`) support an optional `DELAYED` keyword that, if used, returns control to the calling application immediately and actually performs the operation as soon as possible.
- When importing data, turn off autocommit. You may also want to drop indexes (including `FULLTEXT` indexes) and then re-create them after the import has completed.
- Database tables must be indexed to improve the performance of data retrieval. Determining what to index is not a trivial task and involves analyzing the `SELECT` statements that have been used to find recurring `WHERE` and `ORDER BY` clauses. If a simple `WHERE` clause is taking too long to return results, you can bet that the column (or columns) being used is a good candidate for indexing.
- Have a series of complex `OR` conditions in your `SELECT` statement? You might see a significant performance improvement by using multiple `SELECT` statements and `UNION` statements to connect them.
- Indexes improve the performance of data retrieval but hurt the performance of data insertion, deletion, and updates. If you have tables that collect data and are not often searched, don't index them until it's really necessary. (Indexes can be added and dropped as needed.)
- `LIKE` is slow. As a general rule, you are better off using `FULLTEXT` rather than `LIKE`.
- Databases are living entities. A well-optimized set of tables might not be so well optimized after a while. As table usage and contents change, so might the ideal optimization and configuration.
- The most important rule is simply that every rule will be broken at some point.

Tip

Browse the Documentation The official MySQL documentation at http://dev.mysql.com/doc/ is full of useful tips and tricks (and even user-provided comments and feedback). Be sure to check out this invaluable resource.

Summary

In this chapter, you reviewed some important tips and notes pertaining to MySQL performance. Of course, this is just the tip of the iceberg. Now that you have finished reading this book, you are encouraged to experiment and learn as you best see fit.

A

Getting Started with MySQL

This appendix gives you what you need to get started with MySQL.

What You Need

To start using MySQL and to follow along with the chapters in this book, you need access to a MySQL server and copies of client applications (software used to access the server).

You do not need your own installed copy of MySQL, but you do need access to a server. You basically have two options:

- Access to an existing MySQL server, perhaps one provided by your hosting company or place of business or school. To use this server, you need to be granted a server account (a login name and password).
- You may download and install a free copy of the MySQL server for installation on your own computer. (MySQL runs on all major platforms, including Windows, Linux, and macOS.)

Tip

If You Can, Install a Local Server For complete control, including access to commands and features that you will probably not be granted if you are using someone else's MySQL server, install your own local server. Even if you don't end up using your local server as your final production DBMS, you'll still benefit from having complete and unfettered access to all that the server has to offer.

Regardless of whether you use a local server, you need client software (the program you use to actually run MySQL commands). The most readily available client software is the `mysql` command-line utility, which is included with every MySQL installation. You should also install the official MySQL UI, MySQL Workbench, which will be the primary tool you use to interact with MySQL.

Obtaining the Software

To learn more about MySQL, go to dev.mysql.com.

To download a copy of the server, go to dev.mysql.com/downloads/. To follow along with this book, it is recommended that you download and install MySQL 5 (or later). The exact download process varies by platform, but it is clearly explained.

MySQL Workbench may not be installed as part of the core MySQL installation. If needed, you can download it from http://dev.mysql.com/downloads/.

Installing the Software

If you are installing a local MySQL server, do so before installing the optional MySQL utilities. The installation procedure varies from platform to platform, but all installations prompt you for needed information, including the following:

- The installation location. The default is usually fine.
- The password for the root user.
- Ports, service or process names, and more. As a rule, use default values if you are unsure what to specify.

Tip

Multiple Copies of the MySQL Server Multiple copies of the MySQL server may be installed on a single machine, as long as each of them uses a different port.

Preparing to Read This Book

After you have installed MySQL, you can read Chapter 3, "Working with MySQL," to see how to log in and log out of the server, as well as how to execute commands.

The chapters in this book all use real MySQL statements and real data. Appendix B, "The Example Tables," describes the example tables used in this book and explains how to obtain and use the table creation and population scripts.

B

The Example Tables

This appendix outlines the tables in this book and their use.

Writing SQL statements requires a good understanding of the underlying database design. If you don't know what information is stored in what table, how tables are related to each other, and the actual breakdown of data within a row, it is impossible to write effective SQL.

You are strongly advised to actually try every example in every chapter in this book. All the chapters use a common set of data files. To assist you in better understanding the examples and to enable you to follow along with the chapters, this appendix describes the tables used, their relationships, and how to obtain them.

Understanding the Example Tables

The tables used throughout this book are part of an order entry system used by an imaginary distributor of paraphernalia that might be needed by your favorite cartoon characters (yes, cartoon characters; learning MySQL doesn't need to be boring). The tables are used to perform several tasks:

- Manage vendors
- Manage product catalogs
- Manage customer lists
- Enter customer orders

Making this all work requires six tables that are closely interconnected as part of a relational database design. The following sections describe these six tables.

Note

Simplified Examples The tables used here are by no means complete. A real-world order entry system would have to keep track of lots of other data that has not been included here (for example, payment and accounting information, shipment tracking). However, these tables demonstrate the kinds of data organization and relationships you will encounter in most real installations. You can apply these techniques and technologies to your own databases.

Table Descriptions

The following sections describe the six tables used in this book, as well as the columns in each one.

Note

Why the Order ? If you are wondering why the six tables are listed in the order they are, it is due to their dependencies. The `products` table is dependent on the `vendors` table, so `vendors` is listed first, and so on.

The `vendors` Table

The `vendors` table stores data on the vendors whose products are sold. Every vendor has a record in this table, and the `vend_id` column is used to match products with vendors.

TABLE B.1 `vendors` Table Columns

Column	Description
`vend_id`	Unique numeric vendor ID
`vend_name`	Vendor name
`vend_address`	Vendor address
`vend_city`	Vendor city
`vend_state`	Vendor state
`vend_zip`	Vendor zip code
`vend_country`	Vendor country

Every table should have primary keys defined. This table, for example, should use `vend_id` as its primary key. `vend_id` is an automatically incremented field.

The `products` Table

The `products` table contains the product catalog, with one product per row. Each product has a unique ID (in the `prod_id` column) and is related to its vendor by `vend_id` (the vendor's unique ID).

TABLE B.2 `products` Table Columns

Column	Description
`prod_id`	Unique product ID
`vend_id`	Product vendor ID (which relates to `vend_id` in the `vendors` table)
`prod_name`	Product name

Column	Description
prod_price	Product price
prod_desc	Product description

Every table should have primary keys defined. This table, for example, should use `prod_id` as its primary key.

To enforce referential integrity, a foreign key should be defined on `vend_id`, relating it to `vend_id` in `vendors`.

The `customers` Table

The `customers` table stores all customer information. Each customer has a unique ID (in the `cust_id` column).

TABLE B.3 `customers` Table Columns

Column	Description
cust_id	Unique numeric customer ID
cust_name	Customer name
cust_address	Customer address
cust_city	Customer city
cust_state	Customer state
cust_zip	Customer zip code
cust_country	Customer country
cust_contact	Customer contact name
cust_email	Customer contact email address

Every table should have primary keys defined. This table, for example, should use `cust_id` as its primary key. `cust_id` is an automatically incrementing field.

The `orders` Table

The `orders` table stores customer orders (but not order details). Each order is uniquely numbered (in the `order_num` column). Orders are associated with the appropriate customers by the `cust_id` column (which relates to the customer's unique ID in the `customers` table).

TABLE B.4 `orders` Table Columns

Column	Description
order_num	Unique order number
order_date	Order date
cust_id	Order customer ID (which relates to `cust_id` in the `customers` table)

Every table should have primary keys defined. This table, for example, should use `order_num` as its primary key. `order_num` is an automatically incrementing field.

To enforce referential integrity, a foreign key should be defined on `cust_id`, relating it to `cust_id` in `customers`.

The `orderitems` Table

The `orderitems` table stores the items included in each order, one row per item per order. For every row in `orders`, there are one or more rows in `orderitems`. Each order item is uniquely identified by the order number plus the order item (first item in order, second item in order, and so on). Order items are associated with their appropriate order by the `order_num` column (which relates to the order's unique ID in `orders`). In addition, each order item contains the product ID of the item (which relates the item to the `products` table).

TABLE B.5 `orderitems` Table Columns

Column	Description
`order_num`	Order number (which relates to `order_num` in the `orders` table)
`order_item`	Order item number (sequential within an order)
`prod_id`	Product ID (which relates to `prod_id` in the `products` table)
`quantity`	Item quantity
`item_price`	Item price

Every table should have primary keys defined. This table, for example, should use `order_num` and `order_item` as its primary keys.

To enforce referential integrity, foreign keys should be defined on `order_num`, relating it to `order_num` in `orders`, and `prod_id`, relating it to `prod_id` in `products`.

The `productnotes` Table

The `productnotes` table stores notes associated with specific products. A product may have no associated notes, or it may have many associated notes.

TABLE B.6 `productnotes` Table Columns

Column	Description
`note_id`	Unique `note id`
`prod_id`	Product ID (which corresponds to `prod_id` in the `products` table)
`note_date`	Date the note was added
`note_text`	Note text

Every table should have primary keys defined. This table, for example, should use `note_id` as its primary key.

The column `note_text` must be indexed for `FULLTEXT` search use.

This table uses full-text searching, so `ENGINE=MyISAM` must be specified.

Creating the Sample Tables

To follow along with the examples in this book, you need a set of populated tables. You can find everything you need to get up and running on this book's web page, at http://www.forta.com/books/9780138223021/.

There are two ways to create the sample tables:

- The simplest way is to use MySQL Data Import. This is a simple interactive process that will create the database and fully populate it (in much the same way you'd backup and restore a database).
- You can create the database manually and then run two SQL scripts to first create and then populate the tables.

Both options are described below. Do not use both; use one or the other. The first option is recommended if you are using MySQL Workbench.

Using Data Import

Note

For MySQL 8 Only The data export files used here are intended for use with MySQL 8 and have not been tested with earlier versions of MySQL.

MySQL enables entire databases to be imported and exported. You can download an export file from the book web page and simply import it as follows:

1. Download the data export file and save it somewhere on your computer.
2. In MySQL Workbench, click on the **Administration** tab in the **Navigator** panel on the left.
3. Select **Data Import/Restore**.
4. When the Data Import screen is displayed, select **Import from Self-Contained File** and then select the data export file you saved in step 1.
5. Click the **Start Import** button to create and fully populate the `crashcourse` database.

The new `crashcourse` database should now be displayed in the Schemas tab in the Navigator panel.

Using SQL Scripts

Note

For MySQL Only The scripts and files used here to create the sample tables are very DBMS specific; they are designed to be used only with MySQL.

The scripts have been tested extensively with MySQL 4.1 through MySQL 8 but have not been tested with earlier versions of MySQL.

The book's web page contains two SQL script files that you can download:

- `create.sql` contains the MySQL statements to create the six database tables (and define all primary keys and foreign key constraints).
- `populate.sql` contains the SQL `INSERT` statements used to populate these tables.

After you have downloaded the scripts, you can use them to create and populate the tables needed to follow along with the chapters in this book. Here are the steps to follow:

1. Create a new datasource. (Do not use any existing datasource, just to be on the safe side.) The simplest way to do this is by using MySQL Workbench (described in Chapter 2, "Introducing MySQL").
2. Either click the **Create a new schema** button (fourth from the left, with a picture of a cylinder and a + on it) or select the **Schemas** tab in **Navigator**, right-click, and select **Create Schema**.
3. Name the new schema `crashcourse`. You can ignore all other fields. Click **Apply** to create the new database.
4. Either double-click the new datasource if you're using MySQL Workbench (it'll appear in bold if in use) or use the `USE` command if using the `mysql` command-line utility.
5. Execute the `create.sql` script by using either MySQL Workbench or `mysql`. If you are using MySQL Workbench, select **File**, **Open SQL Script**, **create.sql** and then click the **Execute** button (which has a yellow lightning bolt on it). If using the `mysql` command-line utility, specify `create.sql` as the source by specifying the full path to the `create.sql` file.
6. Execute the `populate.sql` script by using either MySQL Workbench or `mysql`. Use the same procedure as in step 5 to populate the new tables.

Note

Create and Then Populate You must run the table creation scripts *before* you run the table population scripts. Be sure to check for any error messages returned by these scripts. If the creation scripts fail, you will need to remedy whatever problem exists before continuing with table population.

And now you should be good to go!

C

MySQL Statement Syntax

To help you find the syntax you need when you need it, this appendix lists the syntax for the most frequently used MySQL operations. For each statement, you'll find a brief description followed by the appropriate syntax. For added convenience, you'll also find cross-references to the chapters where specific statements are taught.

When reading statement syntax, remember the following:

- The | symbol is used to indicate one of several options, so `NULL|NOT NULL` means specify either `NULL` or `NOT NULL`.
- Keywords or clauses enclosed in square brackets [like this] are optional.
- Not all MySQL statements are listed, nor is every clause and option listed.

ALTER TABLE

`ALTER TABLE` is used to update the schema of an existing table. To create a new table, use `CREATE TABLE`. See Chapter 21, "Creating and Manipulating Tables," for more information.

► Input

```
ALTER TABLE tablename
(
    ADD column datatype [NULL|NOT NULL] [CONSTRAINTS],
    CHANGE  column columns datatype [NULL|NOT NULL] [CONSTRAINTS],
    DROP column,
    ...
);
```

COMMIT

`COMMIT` is used to write a transaction to a database. See Chapter 26, "Managing Transaction Processing," for more information.

► Input

```
COMMIT;
```

CREATE INDEX

`CREATE INDEX` is used to create an index on one or more columns. See Chapter 21 for more information.

Input

```
CREATE INDEX indexname
ON tablename (column [ASC|DESC], ...);
```

CREATE PROCEDURE

`CREATE PROCEDURE` is used to create a stored procedure. See Chapter 23, "Working with Stored Procedures," for more information.

Input

```
CREATE PROCEDURE procedurename( [parameters] )
BEGIN
...
END;
```

CREATE TABLE

`CREATE TABLE` is used to create a new database table. (To update the schema of an existing table, use `ALTER TABLE`.) See Chapter 21 for more information.

Input

```
CREATE TABLE tablename
(
       Column datatype [NULL|NOT NULL] [CONSTRAINTS],
       Column datatype [NULL|NOT NULL] [CONSTRAINTS],
       ...
);
```

CREATE USER

`CREATE USER` is used to add a new user account to the system. See Chapter 28, "Managing Security," for more information.

Input

```
CREATE USER username[@hostname]
[IDENTIFIED BY [PASSWORD] 'password'];
```

CREATE VIEW

CREATE VIEW is used to create a new view of one or more tables. See Chapter 22, "Using Views," for more information.

Input

```
CREATE [OR REPLACE] VIEW viewname
AS
SELECT ...;
```

DELETE

DELETE deletes one or more rows from a table. See Chapter 20, "Updating and Deleting Data," for more information.

Input

```
DELETE FROM tablename
[WHERE ...];
```

DROP

DROP permanently removes database objects (tables, views, indexes, and so forth). See Chapters 21, 22, 23, 24, "Using Cursors," 26, and 28 for more information.

Input

```
DROP DATABASE|INDEX|PROCEDURE|TABLE|TRIGGER|USER|VIEW
    itemname;
```

INSERT

INSERT adds a single row to a table. See Chapter 19, "Inserting Data," for more information.

Input

```
INSERT INTO tablename [(columns, ...)]
VALUES(values, ...);
```

INSERT SELECT

INSERT SELECT inserts the results of a SELECT into a table. See Chapter 19 for more information.

Input

```
INSERT INTO tablename [(columns, ...)]
SELECT columns, ... FROM tablename, ...
[WHERE ...];
```

ROLLBACK

ROLLBACK is used to undo a transaction block. See Chapter 26 for more information.

Input

```
ROLLBACK [ TO savepointname];
```

SAVEPOINT

SAVEPOINT defines a savepoint for use with a ROLLBACK statement. See Chapter 26 for more information.

Input

```
SAVEPOINT sp1
```

SELECT

SELECT is used to retrieve data from one or more tables (or views). See Chapter 4, "Retrieving Data," Chapter 5, "Sorting Retrieved Data," and Chapter 6, "Filtering Data," for more basic information. (Chapters 4–17 also cover aspects of SELECT.)

Input

```
SELECT columnname, ...
FROM tablename, ...
[WHERE ...]
[UNION ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY ...];
;
```

START TRANSACTION

START TRANSACTION is used to start a new transaction block. See Chapter 26 for more information.

Input

```
START TRANSACTION;
```

UPDATE

UPDATE updates one or more rows in a table. See Chapter 20 for more information.

Input

```
UPDATE tablename
SET columnname = value, ...
[WHERE ...];
```

D

MySQL Datatypes

This appendix explains the different datatypes used in MySQL.

As explained in Chapter 1, "Understanding SQL," datatypes are basically rules that define what data may be stored in a column and how that data is actually stored.

Datatypes are used for several reasons:

- Datatypes enable you to restrict the type of data that can be stored in a column. For example, a numeric datatype column accepts only numeric values.
- Datatypes allow for more efficient storage internally. For example, numbers and datetime values can be stored in a more condensed format than can text strings.
- Datatypes allow for alternate sorting orders. If everything is treated as strings, `1` comes before `10`, which comes before `2`. (Strings are sorted in dictionary sequence, one character at a time starting from the left.) As numeric datatypes, the numbers would be sorted correctly.

When designing tables, pay careful attention to the datatypes being used. Using the wrong datatype can seriously impact your application. Changing the datatypes of existing populated columns is not a trivial task. (In addition, doing so can result in data loss.)

Although this appendix is by no means a complete tutorial on datatypes and how they are to be used, it explains the major MySQL datatype types and what they are used for.

String Datatypes

The most commonly used datatype is the string datatype. This datatype stores a string, such as names, addresses, phone numbers, or zip codes. As listed in Table D.1, there are basically two types of string datatype that you can use: fixed-length strings and variable-length strings.

A fixed-length string is a datatype that is defined to accept a fixed number of characters, and that number is specified when the table is created. For example, you might allow 30 characters in a first name column or 11 characters in a Social Security number column (which is the exact number needed plus the two dashes). Fixed-length columns do not allow more than the specified number of characters. They also allocate storage space for as many characters as specified. So, if the string `Ben` is stored in a 30-character first name field, a full 30 bytes are stored. `CHAR` is an example of a fixed-length string type.

Variable-length strings store text of variable length. Some variable-length datatypes have a defined maximum size. Others are entirely variable. Either way, only the data specified is saved (and no extra data is stored). `TEXT` is an example of a variable-length string type.

If variable-length datatypes are so flexible, why would you ever want to use fixed-length datatypes? The answer is performance. MySQL can sort and manipulate fixed-length columns far more quickly than it can sort variable-length columns. In addition, MySQL does not allow you to index variable-length columns (or the variable portion of a column). This also dramatically affects performance.

Table D.1 String Datatypes

Datatype	Description
`CHAR`	Fixed-length string from 1 to 255 characters long. Its size must be specified at creation time, or MySQL assumes `CHAR(1)`.
`ENUM`	Accepts one of a predefined set of up to 64KB strings.
`LONGTEXT`	Same as `TEXT`, but with a maximum size of 4GB.
`MEDIUMTEXT`	Same as `TEXT`, but with a maximum size of 16KB.
`SET`	Accepts zero or more of a predefined set of up to 64 strings.
`TEXT`	Variable-length text with a maximum size of 64KB.
`TINYTEXT`	Same as `TEXT`, but with a maximum size of 255 bytes.
`VARCHAR`	Same as `CHAR`, but stores just the text. The size is a maximum, not a minimum.

Tip

Use Quotation Marks Regardless of the form of string datatype being used, string values must always be surrounded by quotation marks. (Single quotation marks are often preferred.)

Caution

When Numeric Values Are Not Numeric Values You might think that phone numbers and zip codes should be stored in numeric fields (after all, they store only numeric data), but doing so would not be advisable. If you store the zip code 01234 in a numeric field, the number 1234 will be saved, and you'll actually lose a digit.

The basic rule to follow is this: If the number is a number used in calculations (sums, averages, and so on), it belongs in a numeric datatype column. If it is used as a literal string (that happens to contain only digits), it belongs in a string datatype column.

Numeric Datatypes

Numeric datatypes store numbers. MySQL supports several numeric datatypes, each with a different range of numbers that can be stored in it. Obviously, the larger the supported range, the more storage space needed. In addition, some numeric datatypes support the use of decimal points (and fractional numbers), whereas others support only whole numbers. Table D.2 lists the frequently used MySQL numeric datatypes.

Note

Signed or `UNSIGNED`? All numeric datatypes, with the exception of `BIT` and `BOOLEAN`, can be signed or unsigned. Signed numeric columns can store both positive and negative numbers, and unsigned numeric columns store only positive numbers. Signed is the default, but if you know that you'll not need to store negative values, you can use the `UNSIGNED` keyword, which will allow you to store values twice as large.

Table D.2 Numeric Datatypes

Datatype	Description
`BIGINT`	Integer value that supports numbers from `-9223372036854775808` to `9223372036854775807` (or 0 to `18446744073709551615` if `UNSIGNED`).
`BIT`	Bit field from 1 to 64 bits wide. (Prior to MySQL 5, `BIT` was functionally equivalent to `TINYINT`.)
`BOOLEAN` (or `BOOL`)	Boolean flag, either 0 or 1, used primarily for on/off flags.
`DECIMAL` (or `DEC`)	Floating point value with varying level of precision.
`DOUBLE`	Double-precision floating point value.
`FLOAT`	Single-precision floating point value.
`INT` (or `INTEGER`)	Integer value that supports numbers from `-2147483648` to `2147483647` (or 0 to `4294967295` if `UNSIGNED`).
`MEDIUMINT`	Integer value that supports numbers from `-8388608` to `8388607` (or 0 to `16777215` if `UNSIGNED`).
`REAL`	4-byte floating point value.
`SMALLINT`	Integer value that supports numbers from `-32768` to `32767` (or 0 to `65535` if `UNSIGNED`)
`TINYINT`	Integer value that supports numbers from `-128` to `127` (or 0 to `255` if `UNSIGNED`).

Tip

Don't Use Quotation Marks Unlike strings, numeric values should never be enclosed within quotes.

Tip

Storing Currency There is no special MySQL datatype for currency values. Use `DECIMAL(8,2)` for currency.

Date and Time Datatypes

MySQL uses special datatypes for the storage of date and time values, as listed in Table D.3.

Table D.3 Date and Time Datatypes

Datatype	Description
`DATE`	Date from `1000-01-01` to `9999-12-31` in the format *`YYYY-MM-DD`*.
`DATETIME`	A combination of `DATE` and `TIME`.
`TIMESTAMP`	Functionally equivalent to `DATETIME` (but with a smaller range).
`TIME`	Time in the format *`HH:MM:SS`*.
`YEAR`	A two- or four-digit year; two-digit years support the range 70 (1970) to 69 (2069), and four-digit years support the range 1901 to 2155.

Binary Datatypes

Binary datatypes are used to store all sorts of data (even binary information), such as graphic images, multimedia, and word processor documents (see Table D.4).

Table D.4 Binary Datatypes

Datatype	Description
`BLOB`	Blob with a maximum length of 64KB.
`MEDIUMBLOB`	Blob with a maximum length of 16MB.
`LONGBLOB`	Blob with a maximum length of 4GB.
`TINYBLOB`	Blob with a maximum length of 255 bytes.

Note

Datatypes in Use If you would like to see a real-world example of how different datatypes are used, see the sample table creation scripts described in Appendix B, "The Example Tables."

E

MySQL Reserved Words

This appendix lists the MySQL *keywords*, which are special words used in performing SQL operations. Do not use these keywords when naming databases, tables, columns, and any other database objects. These keywords are considered reserved.

```
ACTION
ADD
ALL
ALTER
ANALYZE
AND
AS
ASC
ASENSITIVE
BEFORE
BETWEEN
BIGINT
BINARY
BIT
BLOB
BOTH
BY
CALL
CASCADE
CASE
CHANGE
```

```
CHAR
CHARACTER
CHECK
COLLATE
COLUMN
CONDITION
CONNECTION
CONSTRAINT
CONTINUE
CONVERT
CREATE
CROSS
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURSOR
DATABASE
DATABASES
DATE
DAY_HOUR
DAY_MICROSECOND
DAY_MINUTE
DAY_SECOND
DEC
DECIMAL
DECLARE
DEFAULT
DELAYED
DELETE
DESC
```

```
DESCRIBE
DETERMINISTIC
DISTINCT
DISTINCTROW
DIV
DOUBLE
DROP
DUAL
EACH
ELSE
ELSEIF
ENCLOSED
ENUM
ESCAPED
EXISTS
EXIT
EXPLAIN
FALSE
FETCH
FLOAT
FOR
FORCE
FOREIGN
FROM
FULLTEXT
GOTO
GRANT
GROUP
HAVING
HIGH_PRIORITY
HOUR_MICROSECOND
```

```
HOUR_MINUTE
HOUR_SECOND
IF
IGNORE
IN
INDEX
INFILE
INNER
INOUT
INSENSITIVE
INSERT
INT
INTEGER
INTERVAL
INTO
IS
ITERATE
JOIN
KEY
KEYS
KILL
LEADING
LEAVE
LEFT
LIKE
LIMIT
LINES
LOAD
LOCALTIME
LOCALTIMESTAMP
LOCK
```

```
LONG
LONGBLOB
LONGTEXT
LOOP
LOW_PRIORITY
MATCH
MEDIUMBLOB
MEDIUMINT
MEDIUMTEXT
MIDDLEINT
MINUTE_MICROSECOND
MINUTE_SECOND
MOD
MODIFIES
NATURAL
NO
NO_WRITE_TO_BINLOG
NOT
NULL
NUMERIC
ON
OPTIMIZE
OPTION
OPTIONALLY
OR
ORDER
OUT
OUTER
OUTFILE
PRECISION
PRIMARY
```

```
PROCEDURE
PURGE
READ
READS
REAL
REFERENCES
REGEXP
RELEASE
RENAME
REPEAT
REPLACE
REQUIRE
RESTRICT
RETURN
REVOKE
RIGHT
RLIKE
SCHEMA
SCHEMAS
SECOND_MICROSECOND
SELECT
SENSITIVE
SEPARATOR
SET
SHOW
SMALLINT
SONAME
SPATIAL
SPECIFIC
SQL
SQL_BIG_RESULT
```

```
SQL_CALC_FOUND_ROWS
SQL_SMALL_RESULT
SQLEXCEPTION
SQLSTATE
SQLWARNING
SSL
STARTING
STRAIGHT_JOIN
TABLE
TERMINATED
TEXT
THEN
TIME
TIMESTAMP
TINYBLOB
TINYINT
TINYTEXT
TO
TRAILING
TRIGGER
TRUE
UNDO
UNION
UNIQUE
UNLOCK
UNSIGNED
UPDATE
USAGE
USE
USING
UTC_DATE
```

```
UTC_TIME
UTC_TIMESTAMP
VALUES
VARBINARY
VARCHAR
VARCHARACTER
VARYING
WHEN
WHERE
WHILE
WITH
WRITE
XOR
YEAR_MONTH
ZEROFILL
```

Index

Symbols

A

B

C

D

E

F

G-H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z

This page intentionally left blank

Register Your Product at informit.com/register

Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's leading learning company. At informit.com, you can

- Shop our books, eBooks, and video training. Most eBooks are DRM-Free and include PDF and EPUB files.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

** Offers subject to change.*

*** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

Connect with InformIT—Visit informit.com/community

twitter.com/informit

Pearson

informIT

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Oracle Press • Peachpit Press • Pearson IT Certification • Que