

LEARN SQL IN A MONTH OF LUNCHES

- Wildcards and null values
- Querying multiple tables
- Filtering and sorting
- Set operators

- Logical operators
- Grouping data
- Using variables
- Converting data

- Manipulating data
- Using string functions
- Using date and time
- Making decisions



JEFF IANNUCCI



MEAP Edition
Manning Early Access Program

Learn SQL in a Month of Lunches
Version 16

Copyright 2024 Manning Publications

For more information on this and other Manning titles go to manning.com.

welcome

Thank you for purchasing the MEAP edition of *Learn SQL in a Month of Lunches*. I hope this early access will provide immediate benefits to you, and that, with your feedback, we can make the final book even more helpful to future readers.

SQL was created in the 1970s, and yet with the ever-increasing use of databases, the demand for professionals who know SQL has continued to rise. However, despite being a programming language, much of the current demand is for SQL knowledge in non-programmer positions. In today's data-driven marketplace it is now business analysts, product managers, marketers, and many others who need direct access to data to make critical business decisions.

This book is designed for all SQL beginners, including those who are not necessarily software developers. It emphasizes practical ways to use SQL, rather than theoretical, helping you quickly assimilate the basics needed to begin implementing what you've learned. If you know how to use Windows or MacOS and have experience using spreadsheet applications, then you absolutely are ready to start learning how to use SQL to read and manipulate data, create database objects, and more.

Each chapter of this book is short enough to complete within an hour, whether that be during lunchtime or some other part of your day. Month of Lunches books are also designed to help you be immediately effective, so by the second chapter you will already have written your first SQL statement and examined its core components. Subsequent chapters will focus on teaching skills that you can apply to real-life scenarios and will include plenty of exercises to help you reinforce what you have learned.

I started working with SQL as a non-programmer decades ago, and to this day, it is still a valuable skill offering huge opportunities for programmers and non-programmers alike. With that in mind, I've written this book to show not only how to use SQL, but also provide tips and warnings that get you up and running quickly with real-world knowledge and skills. My hope is this book not only reveals to you how easy it is to learn SQL, but also that it helps you develop the proficiency to elevate your career for decades.

As noted earlier, I encourage you to leave feedback in the [liveBook Discussion forum](#). Your questions, comments, or suggestions are essential for making this the best book possible, and I sincerely appreciate your recommendations for improvements that will increase your understanding of SQL.

Thank you again for your interest, and for purchasing the MEAP!

—Jeff Iannucci

brief contents

CHAPTERS

- 1 Before you begin*
- 2 Your first SQL query*
- 3 Querying data*
- 4 Sorting, skipping, and commenting data*
- 5 Filtering on specific values*
- 6 Filtering with multiple values, ranges, and exclusions*
- 7 Filtering with wildcards and null values*
- 8 Querying multiple tables*
- 9 Using different kinds of joins*
- 10 Combining queries with set operators*
- 11 Using subqueries and logical operators*
- 12 Aggregating data*
- 13 Using variables*
- 14 Querying with functions*
- 15 Combining or calculating values with functions*
- 16 Inserting data*
- 17 Updating and deleting data*
- 18 Storing data in tables*
- 19 Creating constraints and indexes*
- 20 Reusing queries with views and stored procedures*
- 21 Making decisions in queries*

22 Using cursors

23 Using someone else's script

24 Never the end

1

Before you begin

Nearly every act of our lives generates data. Every purchase we make, every mile we travel, and every internet link we click adds to a colossal and ever-growing amount of data. It's mind-boggling to think how much information is being generated, and how, for many organizations, this data has become the most valued of all assets.

All this data must be stored somewhere, in a place often referred to as "the database." This database can be a virtual folder with any number of files or a sophisticated product designed for performance or scalability. Some people even refer to an Excel spreadsheet as a database, although we'll discuss the validity of that claim in the next chapter.

For many organizations, data needs more than a place to be stored. It needs to be secure, able to grow, and searchable by a multitude of users and applications. To meet all these requirements, much of this prized data is contained in a certain type of database: a *relational* database. Relational databases have been around since the 1970s and have been in use in commercially available products for almost as long, and because they are well suited for most business scenarios, they remain extremely popular among organizations throughout the world.

But this data in relational databases is worth something only if it can be used. So how can we read and change data in relational databases? Presumably, this is the reason you are reading this book, because the most popular answer is Structured Query Language, or, as it is more commonly known, SQL.

1.1 Why SQL matters

Even though much of the data of our modern lives is stored in relational databases from different brands, such as Oracle, Microsoft, and IBM, nearly all of them have something in common: their data is primarily accessed and changed using SQL.

SQL is decidedly different from most programming languages. You can't create a web application in SQL. You can't create a mobile app in SQL. You can do one thing with SQL, and that is read and manipulate data in a relational database. That may seem like a limitation, yet SQL is far and away the most popular language to work with relational databases. This means even if you were a software developer using C# or Java or some other language to build an application, you would still use SQL for the parts of your application that need to read or change data in a relational database.

Now, you may be wondering: is SQL worth investing an entire month of lunches? Be assured, learning this language is without a doubt one of the best skills anyone who uses data can acquire, because perhaps the most amazing aspect of SQL is how durable this skill has proven to be. While many application languages may have a lifespan of only a few years before being replaced by some other language that has become more useful and is in higher demand, SQL has been for decades the standard language for querying relational databases and should continue to be so for the foreseeable future. This means the skills you will learn and develop by reading this book and practicing the exercises it contains can potentially benefit you for your entire career.

1.2 Is this book for you?

There is no shortage of books, videos, courses, or websites that offer to teach you SQL, many of which are designed for an audience with software development experience. They frequently start with the history of a language, move on to a discussion of its many technical concepts, and then follow with chapters grouped into showing what various commands do. While there's nothing inherently wrong with that approach, it ignores the many non-technical folks who need to learn SQL. Folks like me.

Despite using SQL for over two decades now, my career did not begin in software development. My first experience with databases was in a position called data administrator, where I was responsible for importing data from various sources into a relational database. I needed to read that data to validate the success of the import process, and the only way to do that was by learning and using SQL.

Even though I had limited programming experience, I was able to quickly grasp how to use SQL, and I'm confident you can as well. Why? Because SQL was designed to be written like the English language. As you will see throughout this book, if you understand how to write in English, you should have little trouble learning how to use SQL.

1.2.1 The many uses for SQL

Of course, data isn't just for the IT department. For example, if you are a business analyst, you can use SQL to quickly retrieve and analyze the data about operational trends to make smarter business decisions. If you are a marketing professional, you can use SQL to uncover actionable insights about recent ad campaigns to help grow your business. If you work in finance, you can use SQL to retrieve vital data to help your company meet compliance requirements.

All this data is the lifeblood of any modern organization, and success depends on having members of nearly every department possess the skills to use relational data to make critical business decisions. This book is designed to help people like you learn SQL to build the skills to do just that. If your technical experience is limited to working with spreadsheets, then you're at a great starting point to begin using SQL.

Then again, if you are a software developer, database administrator, or data scientist, this book does not exclude you at all. It simply takes a different approach to learning. Whereas most other SQL books begin with terminology and concepts, this book will quickly get you using SQL to solve practical problems, while briefly sharing concepts and defining terminology along the way.

Conversely, this book isn't designed to simply teach you a bunch of SQL commands. Instead, it is designed to progressively show you how to apply components of the SQL language to do your job, regardless of your level of computer programming experience.

1.2.2 The many flavors of SQL

Even though we will be using a MySQL database to learn about SQL, nearly all the SQL concepts and techniques will work with any relational database. This means what you learn will apply to any of the following database management systems:

- IBM DB2
- MariaDB
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL

Once you have developed a solid foundation for using SQL, you can easily work with data in any of these systems. However, be aware there will be occasional exceptions for individual systems, which will be noted throughout this text so that you can be proficient in whatever system you may be using to work with data.

1.3 How to use this book

The idea of this book is that you will read one chapter each day. You don't have to read during lunch, but each chapter should take about 40 minutes to read, leaving you with about 20 minutes to practice what was shown while you finish eating.

1.3.1 The main chapters

Chapters 1 and 2 will help you get up to speed quickly, getting you familiar with not only the idea of a table and how to think about querying it but also the tools we will be using throughout this book. In some ways, this makes them the most important chapters of the book.

Chapters 3 through 22 represent the primary content, so you can expect to complete them in about a month—even a short month like February. Not every chapter will require a full hour, but it's important to follow the order, as each chapter builds upon the skills and commands demonstrated in previous chapters. And, while you are certainly free to read multiple chapters per day, I'd recommend focusing on a single chapter daily and spending ample time practicing what you've learned. Doing this will give your brain time to focus on just a handful of concepts and examples, which should prove to be optimal in quickly solidifying your knowledge. As the great basketball coach John Wooden said, "Be quick, but don't hurry."

1.3.2 Hands-on labs

Nearly all chapters will include a short lab exercise for you to apply the concepts and commands you learned. Don't think of these as quizzes, but rather as your opportunity to apply and reinforce your new SQL skills. Though the answers to these labs will appear at the end of each chapter, I can't stress enough how vital working through these labs will be in retaining your new knowledge.

1.3.3 Further exploration

Since this book is designed for those who are just starting to use SQL, it's only scratching the surface of the ways you can use and manipulate relational data. Because of this, you will see some chapters ending with suggestions for further exploration of ways to use the concepts and commands. If you have the time, and hopefully the inclination, take a look at these resources to expand your ever-growing SQL skill set.

1.4 Setting up your lab environment

Your time is valuable, so let's get started with setting up your lab environment. This won't be resource intensive and can likely be set up on your own computer in just a few minutes. We'll install only two pieces of free software, and then execute some ready-made SQL scripts to give us some data to use.

1.4.1 Installing MySQL and MySQL Workbench

The first step is to download MySQL and install it on the computer of your choice. MySQL is not only freely available but also one of the most popular relational database applications in the world.

We'll also install MySQL Workbench, which is the tool we'll use to execute all the queries contained in this book. It also uses very few resources, so you shouldn't worry about installing it on a laptop.

The steps for downloading and installing both of these can be found at my GitHub repository, located at <https://github.com/desertdba/learn-SQL-in-a-month-of-lunches>. Since the MySQL software is frequently updated, the version numbers you see may be later than the ones shown in the documentation. Don't worry about that, as nothing we do should be affected by newer versions.

1.4.2 Executing the lab scripts

Throughout this book, we'll rely on a single set of data for our queries. The data is based on a set of orders from a hypothetical book publisher of SQL-based novels, using a database named `sqlnovel` for all our queries. We'll discuss this data more throughout the book, but for now, let's just create the database and populate the sample data by executing a prepared SQL script.

The steps for setting up our SQL Novels database are also located at <https://github.com/desertdba/learn-SQL-in-a-month-of-lunches>, and they are even more simple than the process for installing MySQL and MySQL Workbench. Although you are likely to simply execute the script, near the end of the book we will actually go back and review the script to examine what exactly it does. By that point, you should be able to create your own sets of data!

1.5 Online resources

Creating a lab environment for testing may sound intimidating for anyone who hasn't done such a thing, so as a supplement to this book I've created a few videos that show you how to easily do this for both Windows and MacOS. In these videos, I guide you through the entire process described in the previous section.

Throughout the book, I'll give you examples and exercises to try. You are encouraged to type out all scripts on your own, and you may even prefer to write SQL in a different style than presented in this work. However, in typing out the SQL, you may sometimes encounter an error that you don't understand. For this reason, the online resources will also contain every SQL script presented in this book. Please try to use them only for troubleshooting, as typing the actual SQL yourself will help you learn faster than simply copying scripts.

1.6 Being immediately effective with SQL

As with every other book in the *Month of Lunches* series, the primary goal of this book is for you to be immediately effective with SQL. In nearly every chapter that follows, a particular part of the SQL language will be presented and briefly discussed, though the majority of any given chapter will focus on how to apply what you just learned, using real-world scenarios. Furthermore, at the end of every chapter, you will get hands-on practice yourself, with exercises to complete in a lab environment.

As stated earlier, if you are looking for a deep dive into relational database theory and history, many other books out there can guide you down that path. Although there are many points throughout this book where details and nuances are discussed, every chapter is driven by the goal of making you immediately effective at accomplishing actual tasks.

OK, enough about this book. Let's start using SQL!

2

Your first SQL query

Chapter 1 ended with a word about being immediately effective, and so this chapter aims to do just that. We're going to start looking at some data as it might be stored in a relational database, and we're going to examine the way that data is structured. Doing this will help you better understand some terms to describe the data, which we will use throughout the book. Don't worry, though: You'll see just a handful of terms, and they are all words you have seen and used in conversation. We are just defining them in the context of data stored in relational databases.

Additionally, we'll get you started with your first query. In case you didn't know, a *query* refers to any time you execute some SQL to retrieve data. As you can imagine, as you progress through this book, you'll execute quite a lot of queries to level up your SQL skills. If you have not already completed the installations of MySQL and the MySQL Workbench, referred to in Chapter 1, as well as executed the `Create_SQLNovel_database.sql` script to create our sample database, please do this now so that you are ready to begin querying data.

Before we begin querying, let's look at some data.

2.1 You know tables if you already know spreadsheets

Although it's not a prerequisite for learning SQL in this book, it will be helpful if you have experience working with Microsoft Excel or some other spreadsheet program. You may not realize it, but spreadsheets are structured similarly to the most fundamental objects in any database. We'll also introduce a few terms in this section to help make sense of the way data is stored in a relational database, or as it is more accurately known, a Relational Database Management System (RDBMS).

Now, we don't just gather data and dump it into an RDBMS, but rather we organize and store it in objects based on the nature of the data. These objects are known as *tables* and we typically organize data into tables relating to elements, like orders or customers or payments. Tables are the building blocks of any RDBMS and they are structured quite a bit like spreadsheets, so looking at a spreadsheet will help provide a visual example to better understand the associated terms we will discuss in this chapter and throughout the book.

In case you don't know the basic terms used to describe a spreadsheet, let's look at a typical spreadsheet in figure 2.1, containing information about some extraordinary fictional books.

	A	B	C	D	E
1	Title	Price	Advance	Royalty	Publication Date
2	Pride and Predicates	\$ 9.95	\$ 5,000.00	15%	4/30/2015
3	The Join Luck Club	\$ 9.95	\$ 6,000.00	12%	2/6/2016
4	The Catcher in the Try	\$ 8.95	\$ 5,000.00	10%	4/3/2017
5	Anne of Fact Table	\$ 12.95	\$ 10,000.00	15%	1/12/2018
6	The DateTime Machine	\$ 7.95	\$ 5,500.00	15%	2/4/2019
7	The Great GroupBy	\$ 10.95	\$ -	20%	12/23/2019
8	The Call of the While	\$ 8.95	\$ 2,500.00	15%	3/14/2020
9	The Sum Also Rises	\$ 7.95	\$ 5,000.00	12%	11/12/2021
10					

Figure 2.1 A spreadsheet with five columns (A through E) of values for several extraordinary fictional books. The structure is organized similarly to a table in a database.

Consider this a set of data, commonly known as a *data set*. Seems easy enough, right? The data set is stored in a spreadsheet, but if this data were stored in a table, it would essentially have the same structure. I've said I don't want to overload you with jargon, but there are three simple but critical terms relating to tables that need to be understood to get started using SQL to read and manipulate any data contained in tables. And as I noted at the beginning of the chapter, these are likely words you have heard before:

- Column
- Row
- Value

At the most basic level, a table is a construct of one or more *columns* of data. Columns run vertically, like columns in architecture, and in figure 2.2, we see columns for Title, Price, Advance, Royalty, and Publication Date, with Title highlighted. You may see or hear the term *field* used when referring to a column, but *field* isn't an actual term in the SQL language.

	A	B	C	D	E
1	<u>Title</u>	<u>Price</u>	<u>Advance</u>	<u>Royalty</u>	<u>Publication Date</u>
2	Pride and Predicates	\$ 9.95	\$ 5,000.00	15%	5/31/2011
3	The Join Luck Club	\$ 9.95	\$ 6,000.00	12%	9/6/2012
4	The Catcher in the Try	\$ 8.95	\$ 5,000.00	10%	4/3/2013
5	Anne of Fact Table	\$ 12.95	\$ 10,000.00	15%	6/12/2014
6	The DateTime Machine	\$ 7.95	\$ 5,500.00	15%	2/4/2015
7	The Great GroupBy	\$ 10.95	\$ -	20%	12/23/2015
8	The Call of the While	\$ 8.95	\$ 2,500.00	15%	3/14/2017
9	The Sum Also Rises	\$ 7.95	\$ 5,000.00	10%	11/12/2018

Figure 2.2 The spreadsheet of book titles, with the Title column highlighted to show the vertical nature of columns.

Another term we need to consider is *row*, which refers to each horizontal collection of data in the table. Each row represents a single item of whatever the element of the table is, which in this case is the title of a book. In figure 2.3, we can see that each row has the same structure and follows the same order of columns. This is a requirement, as each row must include representation for all columns in a table.

These rows are also enumerated in the left sidebar, but in any given table the designer may or may not have included explicit identifiers for each record. It's worth noting the terms *row* and *record* are often used interchangeably because certain applications refer to rows as *records*, but for tables in most RDBMSs, the correct term is *row*.

	A	B	C	D	E
1	<u>Title</u>	<u>Price</u>	<u>Advance</u>	<u>Royalty</u>	<u>Publication Date</u>
2	Pride and Predicates	\$ 9.95	\$ 5,000.00	15%	5/31/2011
3	The Join Luck Club	\$ 9.95	\$ 6,000.00	12%	9/6/2012
4	The Catcher in the Try	\$ 8.95	\$ 5,000.00	10%	4/3/2013
5	Anne of Fact Table	\$ 12.95	\$ 10,000.00	15%	6/12/2014
6	The DateTime Machine	\$ 7.95	\$ 5,500.00	15%	2/4/2015
7	The Great GroupBy	\$ 10.95	\$ -	20%	12/23/2015
8	The Call of the While	\$ 8.95	\$ 2,500.00	15%	3/14/2017
9	The Sum Also Rises	\$ 7.95	\$ 5,000.00	10%	11/12/2018

Figure 2.3 The spreadsheet of books, with the first horizontal collection of data highlighted to show the horizontal nature of rows.

The last term, at least for now, is *value*, which represents the distinct pieces of information described by the columns of the data set, and every row contains one value for each column. For example, in figure 2.4 the value for Title in the last row in our data set is The Sum Also Rises, and the value for Price of that row is \$7.95. It's worth noting that even though all columns are required for all rows, the values for the columns can be empty, such as the Advance value for the row with the Title value of The Great GroupBy.

	A	B	C	D	E
1	Title	Price	Advance	Royalty	Publication Date
2	Pride and Predicates	\$ 9.95	\$ 5,000.00	15%	5/31/2011
3	The Join Luck Club	\$ 9.95	\$ 6,000.00	12%	9/6/2012
4	The Catcher in the Try	\$ 8.95	\$ 5,000.00	10%	4/3/2013
5	Anne of Fact Table	\$ 12.95	\$ 10,000.00	15%	6/12/2014
6	The DateTime Machine	\$ 7.95	\$ 5,500.00	15%	2/4/2015
7	The Great GroupBy	\$ 10.95	\$ -	20%	12/23/2015
8	The Call of the While	\$ 8.95	\$ 2,500.00	15%	3/14/2017
9	The Sum Also Rises	\$ 7.95	\$ 5,000.00	10%	11/12/2018

Figure 2.4 The value of Price in the row with the title The Sum Also Rises highlighted to indicate a value. This is just one value of many.

OK, that's enough terminology about tables for now. Let's start talking about how to use this information to query a table.

2.2 Learning SQL is like taking an English class

A common question many people wonder about is how to pronounce *SQL*, since some folks say "ess-que-ell" while others say "sequel." Considering that the earliest version of SQL was called Structured English Query Language and was abbreviated as SEQUEL, you can see how the latter pronunciation became commonplace. For what it's worth, there was already a trademark on SEQUEL, so the creators dropped the word *English* from the name and shortened the abbreviation to *SQL*.

This brings us to another reason for the popularity of SQL: unlike many other programming languages, it's designed to resemble the English language. You see, SQL is a *declarative* language, meaning you specify what data you want and not how you want to get it, which is something the RDBMS you are using will figure out.

We can take this concept of SQL being a declarative language a step further, using simple verbal declarations to say what we want to do with the data. This may seem unusual, but you will soon see how many basic SQL statements can be similar to a verbal declaration for a simple request. Let's walk through an example of how this works.

Let's suppose you have a table of vegetables named "vegetables" like the one shown in figure 2.5, and you want to know the names of all the vegetables.

<u>VegetableID</u>	<u>Name</u>
1	Artichoke
2	Asparagus
3	Beet
4	Bok Choy
5	Broccoli

Figure 2.5 A vegetables table with two columns and five rows. We’re going to learn how to create a query that shows the names of all the vegetables.

If you want to verbally declare this out loud, you might say something like this:

“I would like all the names of the vegetables.”

SQL isn’t intuitive enough for that to work, but it isn’t too far off either. To accomplish this hypothetical query, we need to include in your declaration the two most basic keywords used in SQL. A *keyword* is any word in the SQL language that helps you do, well, anything. The first keyword to learn is `SELECT`, which, when it comes to databases, will be your new best friend. Believe me—you and `SELECT` will work together a lot. Why? Simply put, `SELECT` is the keyword used to define what you want to see and how you want to see it.

“I would like to `SELECT` all the names of the vegetables.”

All right, we are on our way to forming an actual SQL query, but we need to add something else, which is the `FROM` keyword. The `FROM` keyword specifies which data set we want to look at, which in this case is the `vegetables` table.

“I would like to `SELECT` all the names `FROM` the `vegetables` table.”

That’s better, but when we specify a data set using `FROM`, we don’t explicitly say it is a table. Even though tables are actually one of several kinds of data sets you can query, your RDBMS can determine the kind of data set based on the name of the data set.

“I would like to `SELECT` all the names `FROM` `vegetables`.”

We’re getting closer. Now let’s consider “`SELECT` all the names” for a moment. If we want to select all the names, then we are in good shape because that is the default for this type of query. Nothing here specifies that we want any particular names, so we don’t need to state that we want them all.

“I would like to `SELECT` names `FROM` `vegetables`.”

Now this part is tricky because we’ll need to look at our table in figure 2.5. We can see that the name of the column with the data we want is called `Name`, not `Names`. As you query more data, you’ll probably find that a column name is hardly ever a pluralized word, since the value in each column rarely has more than one value for each row. Let’s adjust our verbal declaration just a bit:

“I would like to `SELECT` `Name` `FROM` `vegetables`.”

We’re almost there. The last modification is to remove the “I would like to . . .” text since we start queries with a keyword, which in this case is `SELECT`. Also, that part is kind of wordy, don’t you think?

Oh, and one more thing: we need to add a semicolon to the end of the declaration. This tells the RDBMS that this is the end of what we are declaring and that anything else after that is another query:

```
SELECT Name FROM vegetables;
```

There you go! This is the correct way to query the names of all the vegetables.

Now, let's go from designing a query for hypothetical data to writing and executing a query on actual data.

2.3 Writing your first SQL query

For the first bit of actual SQL you are going to write and execute, let's simply seek the outcome of your first query. As we noted previously, we can start by declaring a sentence in English that defines what we want:

"I would like the outcome of my first query"

Fortunately for us, the actual data to be queried already exists in a table named `MyFirstQuery`, and the values are in a column named `Outcome`. How convenient, right? Using what we learned about SQL syntax in the previous section, we can easily craft a simple query to accomplish our goal:

```
SELECT Outcome from MyFirstQuery;
```

As you can see, we ended our query with a semicolon. To add a little more to what we said earlier, in SQL a semicolon is used as a *statement terminator*. We don't need to go deep into this subject. Just know this effectively means we're done with this statement and anything that comes after the semicolon will be another SQL statement. Doing this prevents confusion for the database engine, especially as we get into more complicated statements later, so we will use semicolons as statement terminators in all our SQL queries.

You may wonder what the difference is between a *statement* and a *query*, and whether these terms are interchangeable. Well, statements and queries aren't exactly the same thing, but they are related. Think of a query as a special kind of statement for retrieving data, and as you advance your SQL skills, you will find yourself using other kinds of statements beyond queries. For now, queries are the only kind of SQL statement we will use.

NOTE Depending on which RDBMS you are using, semicolons may or may not be required as a statement terminator. Although it may not be required for an RDBMS you find yourself using, it's good to start developing the habit of ending all SQL statements with a semicolon for when it will be required – even statements as simple as your first SQL statement.

All right, before we go any further into the weeds about statements, let's get back to our query. Now that we have our query, the next thing we need to do is to open MySQL Workbench so that we can execute the query. *Executing* basically means pressing Send on your instructions to the RDBMS. From there it will figure out how best to complete your query and then send you back the results. Those results will be displayed in a different window in MySQL Workbench.

TRY IT NOW

Open MySQL Workbench and then click to open the **Month of Lunches** connection we created in Chapter 1. Alternatively, you can right-click and select **Open Connection** from the pop-up menu. You should now see something like what is shown in figure 2.6, with **Query 1** highlighted. That represents the top of the Query panel, including the number **1** in the panel. That number indicates the first line of any query we enter here.

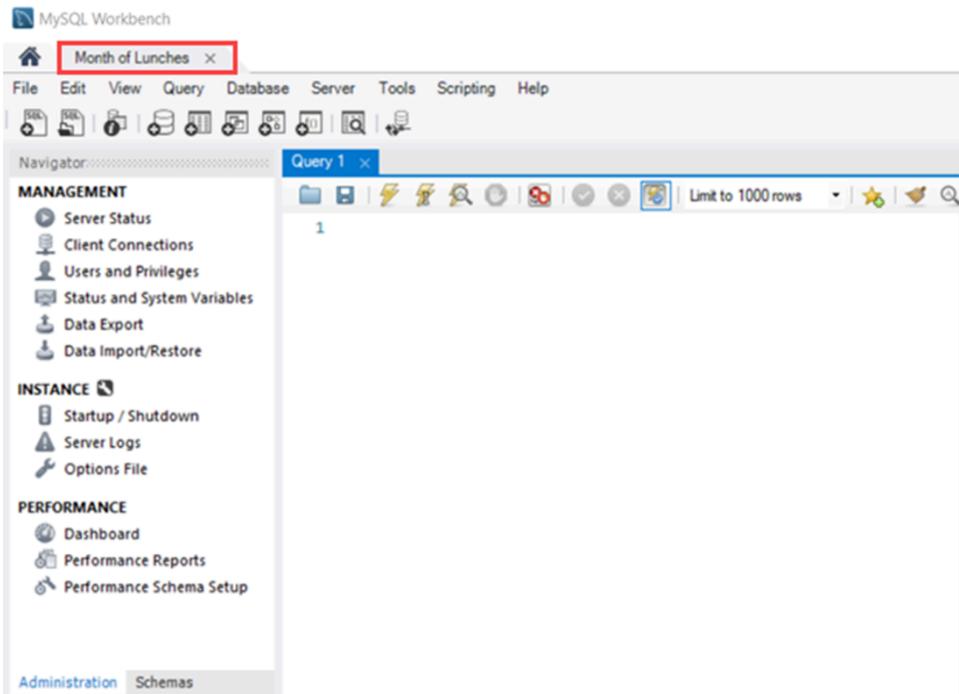


Figure 2.6 The MySQL Workbench open with our Month of Lunches connection, with administration information shown in the Navigator panel. We enter queries like the one we just wrote in the Query panel.

I'd like to point out a few things in MySQL Workbench. The first item is the tab at the top that says Month of Lunches. This tells us the context of our connection, which we set up in Chapter 1 to use the lunch user. We're not going to change that context for any exercise right now, but if you find yourself working with MySQL beyond the scope of this book, you'll always want to pay attention to the connection you are using when querying data.

The second item I want to focus on is the left side. You can see we have an Administration panel, but another tab next to that one, named Schemas, is more important to us. Click the word Schemas and notice the sqlnovel database here. This was created in the scripts we executed in Chapter 1, and we want to set it as the default database for all our queries. To do this, right-click on sqlnovel and select Set as Default Schema from the pop-up menu. Your MySQL Workbench should now look like the one we see in figure 2.7.

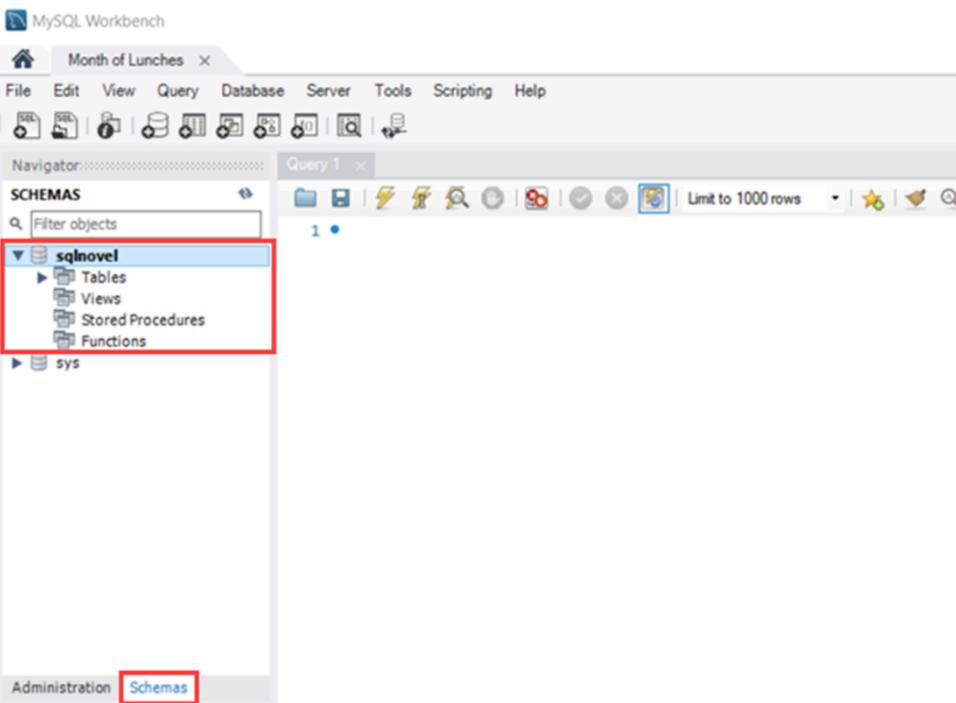


Figure 2.7 The Month of Lunches connection again, but now we see the Schemas information in the Navigator panel. The **sqlnovel** database is now shown in bold text, indicating it is the default database.

As we progress through the book, we'll take time to point out more information that is contained in the Workbench. For now, though, verifying the connection and the database we will be using is enough.

OK, let's get back to the query:

```
SELECT Outcome FROM MyFirstQuery;
```

TRY IT NOW

Move the cursor to the Query panel and click to the right of the 1 and the blue dot. Let's enter your first SQL query here, by typing the query preceding this paragraph. It should look like the image in figure 2.8.

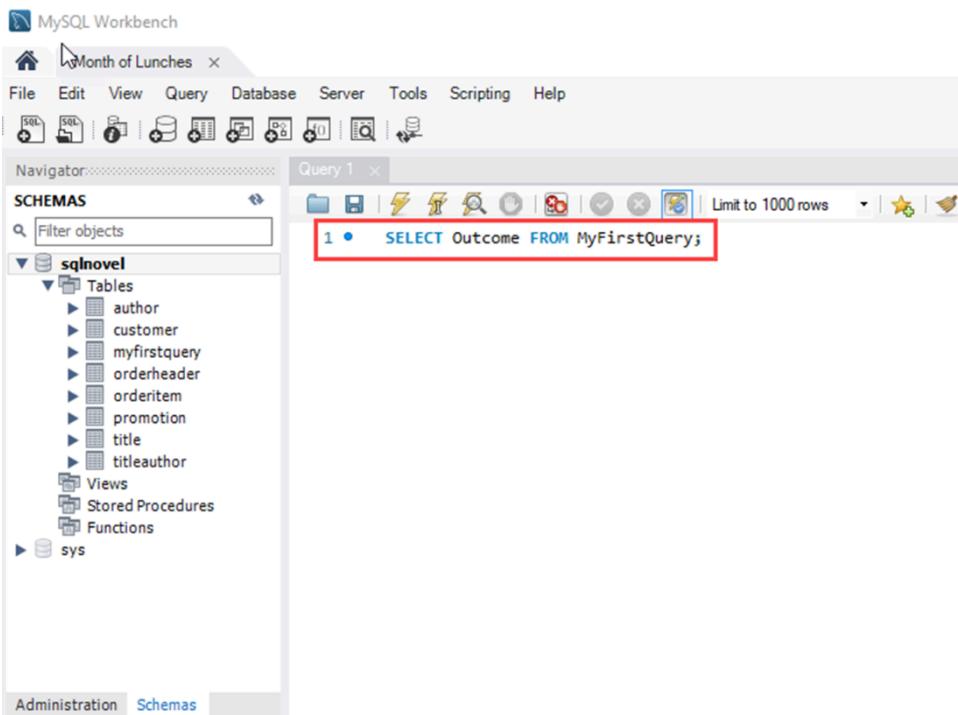


Figure 2.8 The query entered into the Query panel, ready to execute.

I know your anticipation is building, so let me assure you we're almost done. We can execute the query in one of several ways. First, we can select one of the choices from the Query menu at the top. We can choose either **Execute All** or **Execute Current Statement**, which for this single query will do the same thing. This is because we have only one statement in the Query panel.

As you may have noticed on the menu, there are a few hotkeys we could use to execute the query. Pressing Ctrl+Enter on your keyboard will execute the part we have selected, or pressing Ctrl+Shift+Enter will execute the contents of the entire panel. Again, because we have only one line, these will effectively do the same thing for this particular query.

Finally, you may notice some buttons immediately above the first line of the Query panel. Several of them look like lightning bolts, but let's just focus on the first one on the left. That plain lightning bolt, as shown in figure 2.9, will do the same thing as pressing Ctrl+Enter, which is to execute the selected part of the Query panel.

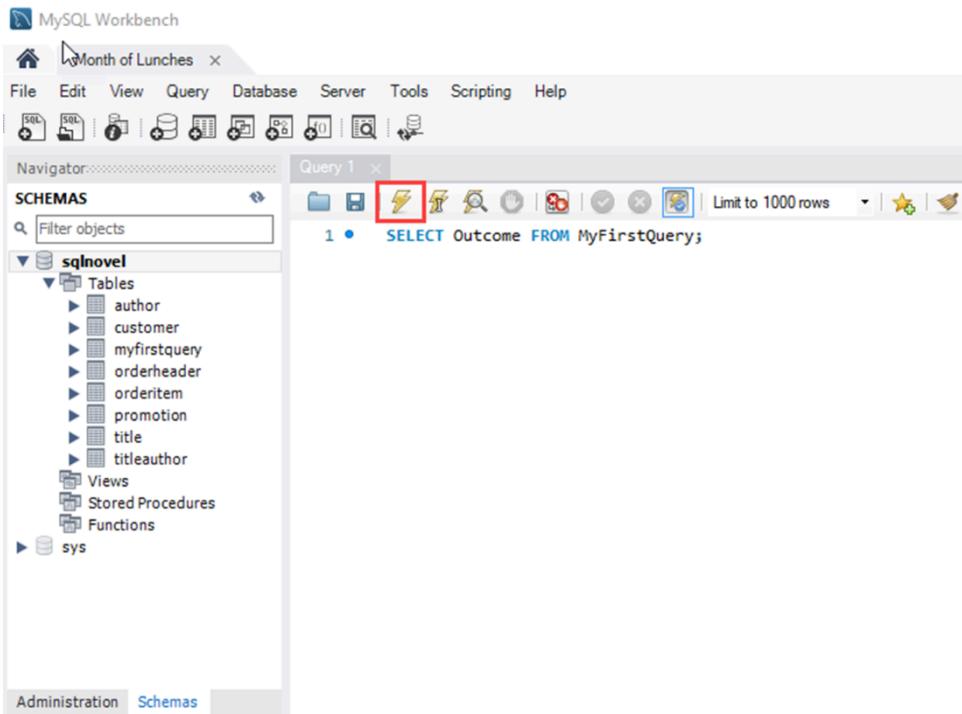


Figure 2.9 The highlighted plain lightning bolt in the Query panel. Clicking the plain lightning bolt will execute the selected part of the Query panel, which is the same thing that pressing **Ctrl + Enter** on your keyboard will do.

TRY IT NOW

Make sure the cursor is placed at the end of the SQL statement, and then choose your method of execution. Go ahead—do it!

After executing the query, you should now see information in two new panels beneath the query, with the result looking like what we see in figure 2.10.

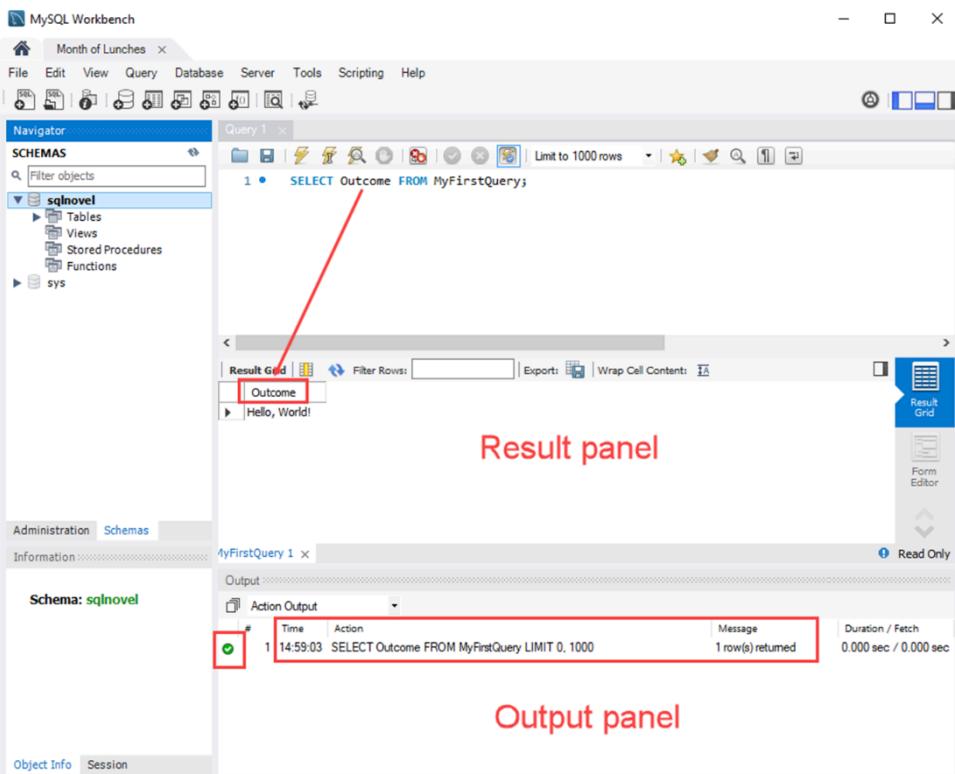


Figure 2.10 The results after query execution. In the Result panel, we can see the result is, “Hello, World!” In the Output panel, a green circle with a check mark indicates the query executed successfully, and other information shows the time it was executed, what the query was, how many rows were returned, and the duration of the query execution.

Immediately beneath the Query panel is the Result panel, and as you can see, the result of the first query is “Hello, World!” If you aren’t a computer programmer, you may not know that one of the traditions of learning a computing language is to first learn how to get “Hello world!” as an output in that language. SQL may not be like most programming languages, but we’re still going to be respectful of traditions.

Take a look at what is immediately above “Hello, World!” though. It’s the word *Outcome*, and this indicates the column that we selected to query. This result, as small as it is, is still considered a data set. It’s one column, and it’s one row, but it’s not a table. We queried a table named `MyFirstQuery`, but the result of the query is a separate data set all by itself.

One final thing about our execution is to notice the Output panel beneath the Result panel. This indicates several bits of information, such as the time our query was executed, what the query was, how many rows were returned, and the duration of the execution of the query. Most importantly, there is a green circle with a check mark, which indicates the query executed successfully. If our query didn’t execute successfully, then we would have seen a red circle with an X to indicate an error. Hopefully, you won’t see too many of those red circles as you work through this book.

2.4 Key terms and keywords

In this chapter, we've covered a handful of simple terms and a few keywords to get you quickly on your way to querying data. Because these concepts and commands are the fundamental building blocks of SQL, let's take a moment to review them.

DATA SET

A *data set* is a logical grouping of data, which can be contained in a database, a spreadsheet, or in any other number of places. As far as we're concerned, we're going to be using SQL to query data in an RDBMS.

TABLE

A *table* is a logical construct of columns that contains a data set. It is the most fundamental way to store data sets in an RDBMS.

COLUMN

A *column* is the vertical grouping of attributes for every row in a table.

ROW

A *row* is the representation of related data in a table.

VALUE

A *value* represents the actual data for a column in a row.

STATEMENT

A *statement* is a way we declare to the RDBMS that we want to do something.

QUERY

A *query* is a special kind of statement that is used to retrieve data.

SELECT

The `SELECT` keyword is used to start queries. The next few chapters of this book will delve deeper into the ways we can use the `SELECT` keyword.

FROM

The `FROM` keyword is used to identify the data set that we want to query.

SEMICOLON

Oh, and don't forget to end your queries with a *semicolon*!

2.5 Lab

Throughout the rest of the book, we will end each chapter with one or more lab exercises to put into practice what we have learned. These exercises are intended to emulate the practical usage of SQL, simulating scenarios you would encounter using other people's data. Since this is the first lab, though, we're going to start off easy with a few simple mental exercises to get you a little more familiar with tables and their structure.

Considering what we've learned using the second SELECT statement, or even considering the example in figure 2.1 as a table, take a moment to consider the following conceptual questions:

1. Imagine the spreadsheet shown in figure 2.1 as a table with several columns. Also, note that the table used in our second SELECT has at least one column. Is it possible for a table to have no columns?
2. Now imagine either of these same data sets as having no rows containing data. Is it possible for a table to have zero rows?
3. In figure 2.1, one of the values for the Advance column does not have a value. If this were a table, do you think it would be required to have a value?
4. Assuming we have a vegetable table in our sqlnovel database that has a column named Name, what do you think would happen if we combined the two queries we discussed in this chapter and executed them at the same time? Think about the result of a query such as the following:

```
SELECT Name from vegetable;
SELECT Outcome FROM MyFirstQuery;
```

Do you think this SQL would execute successfully?

2.6 Lab answers

1. No, it is not possible to have a table with no columns. This is why I mentioned earlier that you could think of a table as a collection of columns. There must always be at least one column; otherwise, there is nowhere to put the values of data.
2. Yes, you can definitely have tables without rows. In fact, every time a table is created, it initially starts with no rows. This is not uncommon, and it's something to consider when querying and the result is zero rows.

3. This is a bit of a tricky question, as it depends on the way the table was set up. We can definitely have rows of valid data that don't have values for particular columns. For example, think of a column for Middle Name in a table of persons. Not everyone has a middle name, so we have to be able to accommodate that lack of data for rows with people who have no middle name. That said, designers can also put restrictions on tables to require all rows to have a value for particular columns, such as a column capturing a last name for all rows in that same table of persons.
4. This actually will execute, and will result in two separate data sets being returned. This is the very reason we put those semicolons at the end of our queries, so that the RDBMS knows we have to separate the queries we are executing.

All right, let's move on to Chapter 3 and learn more fun ways to query using the SELECT keyword!

3

Querying data

In the previous chapter, we looked at a spreadsheet of fictional books to better understand some core concepts about tables in an RDBMS. In this chapter, we're going to work with a table that looks a lot like that spreadsheet and see some of the different ways we can retrieve data using the SELECT command.

Before we do that though, let's take a deeper look at your first query. Although it was very simple, it had all the minimum components for a query. Let's briefly examine those requirements, as well as examine some potential issues regarding formatting and the usage of certain words.

3.1 Rules for the SELECT statement

We've already discussed the conversational way to think of writing a query, so now let's take a moment to consider the technical aspects and requirements as well. Recall your first query, which looked like this:

```
SELECT Outcome FROM MyFirstQuery;
```

In this statement we have four components, each represented by a single word. Technically, the semicolon is a component as well, serving as the statement terminator, but we've already covered how that may or may not be required.

3.1.1 SELECT requirements

The words "Outcome" and "MyFirstQuery" reflect the data we want to select. These are crucial because they help meet the minimum requirements that we need to tell the database in order to retrieve data from a table. These requirements are:

- What data is to be selected
- Where the data is to be selected from

In this case the data to be selected is the “Outcome” column, and the location where the data is to be selected from is the “MyFirstQuery” table. Both of those words follow specific keywords that are included in the SQL catalog of commands.

Those specific keywords are of course SELECT and FROM, and they each represent a *clause* in your SQL statement. All SQL statements are made up of various clauses, but to retrieve data from a table, we are required to use at least these two. We commonly refer to clauses by the keywords used in them, so these would be called the SELECT clause and the FROM clause.

TIP We will always identify the data we want to select immediately after the SELECT keyword, and we will always indicate the location from where we want to select data after the FROM keyword.

It is important to note the order of usage for these clauses in SQL. Throughout the book we will learn several more clauses, and they must always be used in a particular order for a query to execute. For example, we are unable to successfully switch the order of clauses used in your first query, like this:

```
FROM MyFirstQuery SELECT Outcome;
```

Attempting to execute this query will result in another syntax error because the SELECT clause must always come before the FROM clause.

3.1.2 Keywords and reserved words

Keywords such as SELECT and FROM are a subset of *reserved words* of the SQL language used by each RDBMS. When whatever RDBMS you are using finds those reserved words in a query, it presumes you want it to complete a specific action associated with the reserved word. There are numerous reserved words that are universal, but there are also some reserved words specific to the RDBMS you are using.

As you progress in your SQL knowledge, take note of reserved words you use as commands so that you know not to use them as table names or column names.

TIP If you want to know all of the reserved words, you can find them in the documentation on the site where you downloaded MySQL and the Workbench. Every major RDBMS will have documentation online that catalogs their specific reserved words. As a general rule, if you are working in a development interface like the Workbench, you’ll see reserved words show up in a different color than the rest of your SQL.

Using reserved words for object names will result in avoidable headaches from syntax errors, since the RDBMS will be confused about what you want to do. For example, imagine if the MyFirstQuery table had a column named “Select” and you wanted to execute the following query:

```
SELECT Select FROM MyFirstQuery;
```

TRY IT NOW

Type this SQL in your Month of Lunches connection in MySQL Workbench. You’ll see the word Select is shown in a different color than MyFirstQuery, which is your first clue that Select is a reserved word. As noted, there are dozens or even hundreds of reserved words for each RDBMS, so when you see the color indicated for a reserved word, take caution before executing your query.

If you execute the preceding query, you will get an error message that says, “You have an error in your SQL syntax.” This is because the MySQL database engine saw the reserved word “Select” consecutively, which won’t work because you never said what you wanted to select after *the first time* you said SELECT.

3.1.3 Case insensitivity

While we’re looking at this query that won’t execute, notice that “SELECT” and “Select” are both identified as reserved words, even though they are in a different case. Keywords are not case-sensitive, and as such you can see that each of the following queries will successfully execute and return the same result.

```
SELECT Outcome FROM MyFirstQuery;
Select Outcome From MyFirstQuery;
select Outcome from MyFirstQuery;
SeLeCt Outcome fRoM MyFirstQuery;
```

However, just because you can use any kind of case with your keywords doesn’t mean you should. In fact, when writing reusable SQL, many developers prefer typing keywords in uppercase to make code more readable and therefore easier to debug errors. Because of this, the examples throughout this book will continue to show keywords in uppercase whenever they are used.

WARNING Although SQL keywords can be used without regard to case sensitivity, the information relating to data may be case sensitive, depending on the settings of your RDBMS. Be careful when specifying table, column, or value names in your queries, as they may be case-sensitive.

3.1.4 Formatting and white space

One other thing to note about queries is the flexibility we have when using white space. Your RDBMS doesn't care so much about it, and you can format your query in a nearly infinite number of ways with spaces, tabs, and carriage returns. Your first query was written on a single line, but it would also work the same if it were separated into several lines. As such, all three of the following queries will execute and return the same result.

Query 1

```
SELECT Outcome  
FROM MyFirstQuery;
```

Query 2

```
SELECT  
    Outcome  
FROM  
    MyFirstQuery;
```

Query 3

```
SELECT  
    Outcome  
FROM  
    MyFirstQuery;
```

Although there are no universal best practices with formatting, the best advice I can give you is to just be consistent. The goal of adjusting the format is to make the query more readable, so if you find a particular way of formatting that is easier for you to work with, then use it and stick with it.

I think we've gotten all we can out of your first query. It's time to move on to querying data that might be a bit more comparable to the kind you need to work with.

3.2 Retrieving data from a table

For the rest of this chapter, we're going to be examining the title table in our sqlnovel database. Unlike the MyFirstQuery table, the title table has several columns and multiple rows of data.

Unless you've examined the scripts used to create this database, you probably don't know what the names of the columns are in the title table. Fortunately, we can easily find this information using the Workbench. We can look in the Navigator panel in the upper left and notice the triangles next to sqlnovel and Tables. The triangle next to sqlnovel is pointing down, which indicates it has been expanded. This allows us to see Tables, Views, Stored Procedures, and Functions, as shown in figure 3.1.

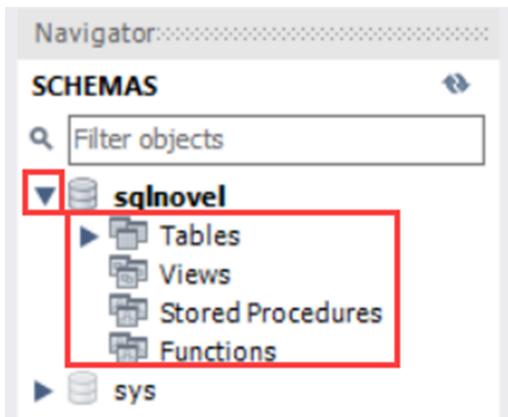


Figure 3.1 The database name has been expanded to show Tables, View, Stored Procedures, and Functions.

The triangle next to Tables is pointing to the right, which means the view of the contents under Tables has been collapsed. To see the columns in the title table or any other table, we need to click that triangle to expand the list of Tables. We'll then need to find the title table and click the triangle next to it to expand further, and then click once more on the triangle next to Columns to expand that as well. Once we have completed those clicks, we can see the names of all columns in the title table, as shown in figure 3.2.

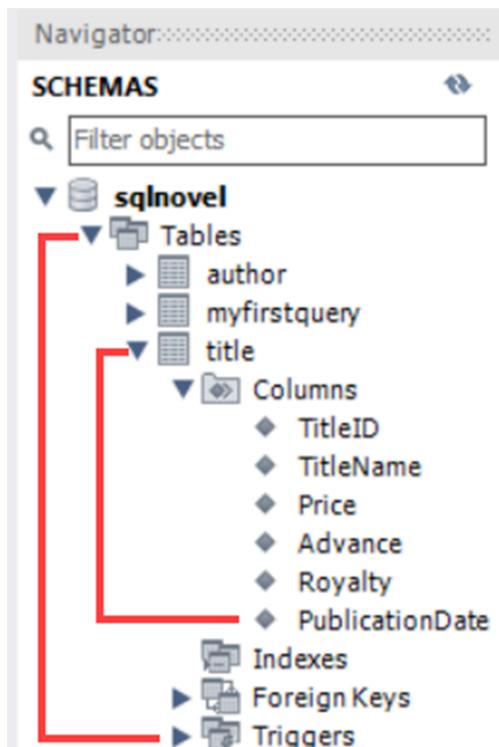


Figure 3.2 The Navigator panel, where Tables has been expanded to show individual table names, and the title table has been expanded to show all columns in that table.

Now that we know the column names, we can start querying the table.

3.2.1 Retrieving an individual column

Let's start with a simple query of the TitleName column from the title table. Because we're going to be increasing the length and complexity of our queries, let's start formatting our queries with the FROM command on a separate line to make it a little more readable.

```
SELECT TitleName
FROM title;
```

TRY IT NOW

Write and execute the preceding query. Consider this your second query (not that anyone is counting).

Executing the query results in the eight rows shown in figure 3.3. If you happen to see the same eight rows but in a different order, don't be alarmed. This is because there is no implicit guarantee of ordering the results of a query.

The screenshot shows a 'Result Grid' window with a header bar containing 'Result Grid', a refresh icon, and a 'Filter Rows:' button. Below the header is a table with one column labeled 'TitleName'. The data rows are:

TitleName
Pride and Predicates
The Join Luck Club
Catcher in the Try
Anne of Fact Tables
The DateTime Machine
The Great GroupBy
The Call of the While
The Sum Also Rises

Figure 3.3 The values of the `TitleName` column in the `title` table are returned in no particular order.

WARNING I'm going to say this again because this is a misconception many SQL users have. There is no implicit guarantee of the order of results of a SQL query. You should not be surprised when you execute the same query at different times and get the same results but in a different order. This can be due to any number of factors, from modifications in the values of the tables being queried to changes in the settings of the server with your database. Remember, SQL is a declarative language, so if your RDBMS isn't explicitly told how to order the results, then the rows can frequently appear in random order. That said, if you peeked ahead in this book, you already know we're going to discuss how to do this in the next chapter when we discuss sorting data.

Something else to take note of is that the name of the column in the query is `TitleName`. You might be wondering why it isn't just `Name` instead. The main reason is that lots of columns in databases contain data with values for names of things or people, and `TitleName` is specific to this table. But another reason relates to what we covered earlier about reserved words. The word "Name" is one of those reserved words.

TRY IT NOW

If you executed the previous query, take a moment now to click near `TitleName` and delete the letters in the `Title` prefix, leaving only `Name`. Did you notice how `Name` is now showing in the same color as `SELECT` and `FROM`? That's because in MySQL the word `Name` is a reserved word.

It's not a keyword command like SELECT or FROM, but it is a reserved word relating to a specific action that is done in this RDBMS. For this reason, the use of the word Name for a column name should be avoided.

TIP Here's a neat trick to save you some typing in the future. Go ahead and delete "Name" from your query as well, leaving a couple of spaces between SELECT and FROM. Now move the cursor to the Navigator panel. Click to select and hold TitleName from the Columns list, move the cursor to the space between SELECT and FROM, and release your click. You should now see TitleName appear as before without having to do any typing. This is a great shortcut to use, especially when dealing with long column names, and it works for other objects such as table names too!

3.2.2 Retrieving multiple columns

Up until now we've only been selecting one column of data, but you will definitely want to write SQL queries that select multiple columns. Let's think about declaring a statement like we did in Chapter 2.

"I would like all the TitleNames and Prices of the titles."

We already know how to convert most of this into a query, so all we need to do now is consider replacing the word "and" with a comma. Since we have multiple column names, let's change the formatting a bit to make multiple column names more readable. Our query will look like this:

```
SELECT
    TitleName,
    Price
FROM title;
```

The comma tells the RDBMS there will be another column requested by our query, just like the word "and" does in the English language. When speaking, we wouldn't end a set of words with "and." For example, we wouldn't say, "I would like all the TitleNames and Prices and . . . of the titles," because the listener would think, ". . . and what?" They would know something else should be included, but it's not clear what that would be. For the same reason, don't put a comma after the last column in your SELECT statement. Doing so will result in a syntax error.

Also note that column order output is completely up to you, the SQL query writer. Just because the columns are in a certain order in the table doesn't mean they can't be rearranged like this:

```
SELECT
    Price,
    TitleName
FROM title;
```

We can even include the same column multiple times if we want, like this:

```
SELECT
    TitleName,
    TitleName,
    Price
FROM title;
```

Having two columns with the same name does introduce a bit of confusion, though. Is there a better way to manage multiple columns with the same name? Why, yes there is!

3.2.3 Renaming output columns using aliases

Although your SELECT query can't change the names of the columns in the tables they are using, they can easily change the name of the output column to whatever you want. Let's declare a statement again.

"I would like all the TitleNames as BookNames of the titles."

Just as you would use the word "as" in your declarative statement, you would use the word AS in your SQL statement to declare the new column name:

```
SELECT
    TitleName AS BookName
FROM title;
```

Your output should now show BookName as the column name instead of TitleName, just as it does in figure 3.4:

	BookName
▶	Pride and Predicates
	The Join Luck Club
	Catcher in the Try
	Anne of Fact Tables
	The DateTime Machine
	The Great GroupBy
	The Call of the While
	The Sum Also Rises

Figure 3.4 The results of the values for `TitleName` are now returned with a column header of `BookName`.

What we have done here is use an *alias*, which is a simple method of renaming the output column to something other than its original column name. We can use column aliases to avoid confusion from similarly titled columns by giving the output columns unique names:

```
SELECT
    TitleName AS BookName,
    TitleName AS AlsoBookName,
    Price
FROM title;
```

Alternatively, you actually don't need to use the word AS to use an alias. You can simply put the *alias name* after the query, although this makes your column aliases a little less obvious:

```
SELECT
    TitleName BookName,
    TitleName AlsoBookName,
    Price
FROM title;
```

Column aliases can be a wonderful tool for making column names more effective. Just remember to avoid using any reserved words as your aliases.

3.2.4 Retrieving all columns

We've seen how to select a single column and multiple columns of data from a table. You will often find you need every column of data in a table to be meaningful, and you will want to retrieve them all. Perhaps you need to write a detailed report that includes the maximum amount of sales data, or maybe you have been asked by auditors to supply every bit of customer data. Whatever the case, there are basically three ways to do this.

The first way is by typing out all the column names, separated by commas. Unless the column names are short and you are very proficient at typing, this is the hardest way.

The second way involves much less typing. You can select the first column in a table in the Navigator panel, click it, and then hold the Shift key and click the last column. You should now see all those columns highlighted, as they are in figure 3.5.

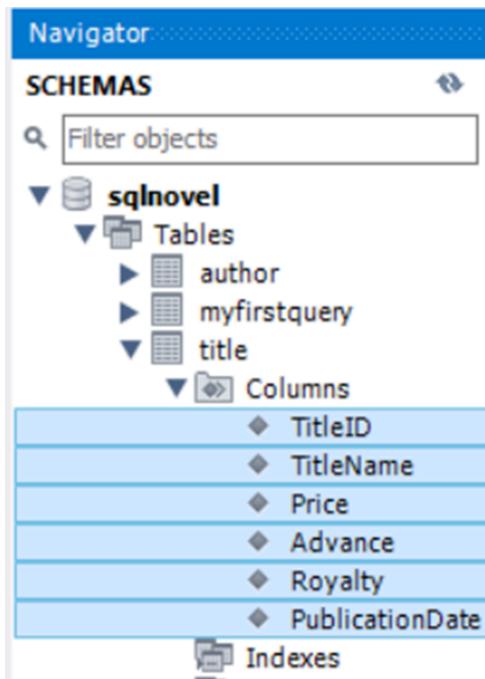


Figure 3.5 The column names for the title table highlighted after selecting the first column name and then holding the Shift key and clicking the last column name.

Now click and hold any of the highlighted columns, drag the cursor between SELECT and FROM in the query panel, and then release. You should now see all the column names listed in your query, as they are in figure 3.6.

The screenshot shows a 'Query 1' window with a toolbar at the top containing various icons. Below the toolbar, the query text is displayed in four numbered lines:

```

1 •  SELECT
2      TitleID, TitleName, Price, Advance, Royalty, PublicationDate
3  FROM title;
4

```

The second line of the query, which contains the column names, is highlighted with a light blue background.

Figure 3.6 shows the columns of the title table after they have been pasted in the Query panel.

TRY IT NOW

Execute this query and you can now see the values for all the columns in the table! If, for whatever reason, the right side of the query is blocked by another panel, you should be able to remove it by selecting View > Panel > Hide Secondary Sidebar from the top menu.

The third method is probably the most common, as it involves less typing and clicking. You can see all columns in a table by using an asterisk in place of column names, commonly referred to verbally as "select star." This is also sometimes referred to as "select all," although that is less common.

```

SELECT
*
FROM title;

```

This will give you the same results as the previous query, and I'm sure you can see why it's so popular. You can easily see the values for all columns with less effort than it takes to type a single column name!

Aside from minimal effort, the other benefit of this method is it will allow us to quickly see all the column names in a table without using the Navigator panel. However, there are two significant reasons to be careful with `SELECT *`.

The first is that you are selecting all the data, which means the RDBMS has to read more data and send it across some network to you as output. It may seem like resources are infinite, but in my experience, the use of `SELECT *` on very large tables uses so much of the system resources that it causes performance problems for other queries. Be very careful when using this method.

The second hazard of `SELECT *` is that since it does not specify any column order, it should never be used in reusable queries. Suppose you write a SQL query for a report that expects five output columns from a table. If one or more columns are later added to or removed from the table, your report may no longer work because there will be a different number of columns.

WARNING For the reasons noted earlier, you should use the SELECT * method only in ad hoc queries, and even then, sparingly. As shown in the second method, it's not that difficult to click and drag the column names we want for a query.

3.3 Lab

1. There is another table in the sqlnovel database named **author**. What two methods could you use to find the names of columns in this table?
2. You need to write a query to return all the first and last names from the author table. How would you write that query?
3. What do you think will happen if we forget to put a comma between column names? Do you think this query will work, and if so, what will be the output?

```
SELECT
    TitleName
    Price
FROM title
```

4. What will happen if we try to use the SELECT * method with an alias, like in the following query?

```
SELECT
    * AS Everything
FROM title;
```

3.4 Lab answers

1. The best method would be to expand the columns folder under the author table in the Navigator panel. If, for some reason, you are using an interface that doesn't allow you to do that, you can use the SELECT * method in the following query.

```
SELECT
    *
FROM author;
```

2. The answer is below:

```
SELECT
    FirstName,
    LastName
FROM author;
```

3. This query will execute, but the results will not be as expected. Since the comma is no longer between TitleName and Price, the word "Price" is now considered a column alias for TitleName. Only one column with an aliased name of Price will be returned, but the values will be from the TitleName column.
4. This query won't execute, as you can't alias column names with SELECT *. If you try this, your first indication of a problem will be the red square with a white X on the line with the alias, as this means the Workbench application has already found your query to have an incorrect syntax.

4

Sorting, skipping, and commenting data

In the previous chapter, we noted that your RDBMS won't return results in a predictable order. This is by design, as any given request may or may not need the data to be ordered, and the RDBMS is simply taking the most efficient way of returning the data requested. It may appear the results are in some sort of order, such as by the order the rows were added to a table, but there are no implicit guarantees for how the data in the results of a query will be ordered.

If we want to be certain of the order of results, we need to explicitly say this in our SQL query. In this chapter we will show how to do that.

There are also some fun features associated with ordering data, which allow us to manipulate the number of rows returned in our result set. This can be useful if you have millions or billions of rows and you need to see only the most recent entries, or the entries with the smallest or largest values.

And since we are still just getting started with writing SQL, we should also discuss how to use *comments* in SQL. These are indispensable tools for you and anyone else who may read your SQL, and if you're going to write queries properly you should start building the habit of using comments today.

4.1 Sorting data

Everywhere we go, we find things sorted in some order. Books in a library are sorted by author name, floors in a building are sorted by number, and events in a daily planner are sorted by date and time. All sorts of things in our lives are organized for ease of use and readability, and the data we use should be no different.

4.1.1 Sorting by one column

To see how we do this in SQL, let's go back to a declarative sentence from the last chapter.

"I would like all the TitleNames and Prices of the titles."

The SQL for this request is as follows:

```
SELECT
    TitleName,
    Price
FROM title;
```

As we can see in figure 4.1, this query won't return the data in a particular way.

TitleName	Price
Pride and Predicates	9.95
The Join Luck Club	9.95
Catcher in the Try	8.95
Anne of Fact Tables	12.95
The DateTime Machine	7.95
The Great GroupBy	10.95
The Call of the White	8.95
The Sum Also Rises	7.95

Figure 4.1 The results of the query of TitleName and Price from the title table. The results are not ordered by either TitleName or Price.

Let's declare that we want to have the same results, but this time ordered by the title name.

"I would like all the TitleNames and Prices of the titles, and I would like the results ordered by TitleName."

As before, the SQL we will use for this is very similar to what we just said. However, to fulfill the new request, we will add a new clause using the ORDER BY keyword.

```
SELECT
    TitleName,
    Price
FROM title
ORDER BY
    TitleName;
```

If we execute this command, we can see in figure 4.2 that the rows returned are the same as before, but now they are ordered as requested.

TitleName	Price
Anne of Fact Tables	12.95
Catcher in the Try	8.95
Pride and Predicates	9.95
The Call of the While	8.95
The DateTime Machine	7.95
The Great GroupBy	10.95
The Join Luck Club	9.95
The Sum Also Rises	7.95

Figure 4.2 The results of the preceding query, with the results now sorted alphabetically by TitleName.

When used, the ORDER BY clause should almost always be the last clause in your SQL statement. With only one exception, which we will get to later in this chapter, specifying any other clauses after the ORDER BY clause in your query will result in a syntax error.

This should be easy to remember if you consider that the ordering of the data will be the last operation the RDBMS will do with your data. This is often a misunderstood point, as there are too many people who use SQL thinking ORDER BY indicates how the data will read. In reality, the RDBMS has to complete the operations for the rest of your SQL statement first, and then, after getting your result set, organize the data in the way your query requests in the ORDER BY clause.

WARNING The additional work that ORDER BY requires of the RDBMS is minimal for the queries in this book since the result sets are only a handful of rows. However, when you start querying sets of data with millions or even billions of rows, adding an ORDER BY can be catastrophic for query performance. Even more problematic is the fact that ordering data for very large data sets can often require large amounts of resources from the computer's processor and memory, resulting in degraded performance for queries other users are running. As your SQL experience grows, always be careful with your use of ORDER BY.

When you consider that your RDBMS has to gather the entire result set before sorting the results, it should make sense that you can even use a column alias in the ORDER BY clause. The column aliases are applied when your result set has been created and before it has been ordered, so SQL logic like the following works as well for sorting data as shown in figure 4.3:

```

SELECT
    TitleName AS NameOfTheBook,
    Price
FROM title
ORDER BY
    NameOfTheBook;

```

NameOfTheBook	Price
Anne of Fact Tables	12.95
Catcher in the Try	8.95
Pride and Predicates	9.95
The Call of the While	8.95
The DateTime Machine	7.95
The Great GroupBy	10.95
The Join Luck Club	9.95
The Sum Also Rises	7.95

Figure 4.3 Using an ORDER BY, the results of the preceding query are now ordered by TitleName, which has been aliased as NameOfTheBook.

4.1.2 Sorting by multiple columns

Of course, ORDER BY isn't limited to the use of a single column. We can use commas in the ORDER BY clause similarly to the way we use them in the SELECT clause to specify ordering by multiple columns.

Consider the following query:

```

SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    Advance,
    Royalty;

```

As figure 4.4 shows, the results of our query are primarily ordered by the Advance column, from lowest to highest. However, when the values in Advance are the same—as they are for the third, fourth, and fifth rows, all with a value of 5000.00—the Royalty column is then used for ordering those three rows.

TitleName	Advance	Royalty
The Great GroupBy	0.00	20.00
The Call of the While	2500.00	15.00
Catcher in the Try	5000.00	10.00
The Sum Also Rises	5000.00	10.00
Pride and Predicates	5000.00	15.00
The DateTime Machine	5500.00	15.00
The Join Luck Club	6000.00	12.00
Anne of Fact Tables	10000.00	15.00

Figure 4.4 The results of a query ordered by the Advance column and then by the Royalty column, both from highest value to lowest.

TRY IT NOW

Now that you've seen a few ways to sort data in the title table, try using ORDER BY as shown in the examples, or sorting on other columns, such as Price or PublicationDate.

4.1.3 Sort direction

When we sort data with ORDER BY, there is an implicit direction of sorting, either alphabetically from A to Z or numerically from lowest to highest value. This is known as *ascending* data order. We can also sort results in the other direction, from Z to A, or from highest to lowest values. This is known as *descending* order, and it must be explicitly stated by adding DESC after the column name in the ORDER BY clause.

Here is an example of the previous query, but with data sorted by Advance values in descending order:

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    Advance DESC,
    Royalty;
```

The results in figure 4.5 show that the results are now ordered by Advance values from largest to smallest. But take a closer look at the fourth, fifth, and sixth rows. They are still sorted in value from largest to smallest Royalty. This is because Royalty is still sorted in ascending order, as no order was specified for this column.

TitleName	Advance	Royalty
Anne of Fact Tables	10000.00	15.00
The Join Luck Club	6000.00	12.00
The DateTime Machine	5500.00	15.00
Catcher in the Try	5000.00	10.00
The Sum Also Rises	5000.00	10.00
Pride and Predicates	5000.00	15.00
The Call of the White	2500.00	15.00
The Great GroupBy	0.00	20.00

Figure 4.5 The results now sorted by Advance descending and Royalty ascending.

For clarity, we can explicitly state the sort order for the Royalty column as well, by adding ASC for ascending sort order to the ORDER BY clause, like this:

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    Advance DESC,
    Royalty ASC;
```

TIP The implicit nature of ascending order in an ORDER BY column can be confusing, so when writing SQL that will be read by others, get into the habit of explicitly stating the direction for ordering, even though you don't need to for ascending. As noted earlier, you should always try to make your SQL as readable as possible.

4.1.4 Sorting by hidden columns

You may encounter certain scenarios where you want to order the results by a column you don't want returned in the result set. This is possible because you can actually order your results by one or more columns that aren't seen. Consider the previous query where we would only return the TitleName, but still order the results by the Advance (descending) and Royalty (ascending) columns.

```

SELECT
    TitleName
FROM title
ORDER BY
    Advance DESC,
    Royalty ASC;

```

The results in figure 4.6 are the same as they were for the previous query, just without the Advance and Royalty columns being returned.

TitleName
Anne of Fact Tables
The Join Luck Club
The DateTime Machine
Catcher in the Try
The Sum Also Rises
Pride and Predicates
The Call of the While
The Great GroupBy

Figure 4.6 shows only the TitleName values in the results, but the rows are still sorted by Advance descending and Royalty ascending.

How can we sort by data that isn't included in our SELECT? The RDBMS accomplishes this little bit of magic by actually adding Advance and Royalty to the result set before it is returned to you, then organizing the data as requested, and finally returning only the columns requested in the SELECT clause. As you can imagine, this is an extra bit of work, so be careful using this technique with very large data sets.

4.1.5 Sorting by position

If column names seem too long to type, there is a quicker way to specify sort order. This can be done by listing the *numerical column position* in the SELECT clause instead of the column name. Think of the numerical order of the columns in the SELECT clause: TitleName (1), Advance (2), and Royalty (3):

```

SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    2 DESC,
    3 ASC;

```

The sort order is now listed as Advance descending and Royalty ascending. We know this because we can determine that Advance is the second column, represented by the 2 value in the ORDER BY, and Royalty is the third column, represented by the 3 value.

WARNING This kind of shorthand notation in the ORDER BY may be useful when you are quickly writing SQL for ad hoc queries, but because the readability is inferior to explicitly naming columns to be sorted, it should be avoided in any reusable SQL that you write. As you can imagine, if the columns in the SELECT change, then ordering by position would be sorted by completely different columns.

4.2 Skipping data

The result set of every query we've run includes all the data in the table. What if you didn't want all the data returned? There will be occasions when you want only a handful of rows to survey, or maybe even just one, skipping most of the result set. For example, you may need to look at a table of data you are unfamiliar with and you just want to see how the data is formatted. We can certainly do this in SQL.

4.2.1 Using LIMIT to reduce results

Let's use our declarative English language first, to state our intentions of finding just three published books.

"I would like all the TitleNames and PublicationDates, but limit the results to the three rows."

The keyword here is LIMIT, as that will be used to reduce the result set to a specified number of rows. We can accomplish this by using LIMIT like this:

```

SELECT
    TitleName,
    PublicationDate
FROM title
LIMIT 3;

```

The RDBMS will grab the first 3 rows it can find, so your results should look something like figure 4.7.

TitleName	PublicationDate
Pride and Predicates	2011-05-31 00:00:00
The Join Luck Club	2012-09-06 00:00:00
Catcher in the Try	2013-04-03 00:00:00

Figure 4.7 The results of using LIMIT to return only three rows.

Using LIMIT with a result set now returns only three rows instead of all eight in the table. Although this may not seem like a useful command, it can be incredibly helpful if you quickly want to sample the column names and types of values they contain.

TRY IT NOW

Using SELECT * and LIMIT, write a query that allows you to quickly sample some rows in the title table. You may or may not get rows with the same three titles shown in figure 4.7, but that's because we didn't specify any sort order.

Since the previous query would be used to sample data, the results can be imprecise. Let's use our declarative English language first, to state our intentions of finding something more precise—namely, the three most recently published books.

"I would like all the TitleNames and PublicationDates, but limit the results to the three most recent PublicationDates."

To accomplish this query, we will bring back ORDER BY, sorting by PublicationDate descending to give us rows with the most recent (latest) date values.

```
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3;
```

Note the order of the clauses here, as the LIMIT clause is after the ORDER BY. The LIMIT clause is the only clause that should ever follow the ORDER BY clause, and if included will always be the last clause in a SQL query. Placing the LIMIT clause anywhere else will result in a syntax error.

The results of this query are shown in figure 4.8, with the three most recent TitleName values with their PublicationDate values being returned.

TitleName	PublicationDate
The Sum Also Rises	2018-11-12 00:00:00
The Call of the White	2017-03-14 00:00:00
The Great GroupBy	2015-12-23 00:00:00

Figure 4.8 The three most recently published TitleNames and their PublicationDates when using ORDER BY on PublicationDate and LIMIT.

TIP Although it is not required, you will almost always want to use ORDER BY whenever you use the LIMIT clause. Why? Because you will likely intend to read a limited sample of rows based on being the oldest, newest, largest, smallest, or some other order. As noted, using LIMIT without specifying the order will return random and unpredictable results.

4.2.2 Using OFFSET to select a different limited set

The scenario of writing a SQL statement to find the most recent data is not uncommon, but you may also find times when you want to skip certain rows other than the most or least. In those cases, you have another feature of the LIMIT clause you can use.

The way to do this is by using an additional option in the LIMIT clause, OFFSET. OFFSET cannot be executed without LIMIT, but it can direct the RDBMS to ignore a specified number of rows before it starts returning the rows indicated by the LIMIT clause.

Let's take the previous query but skip the first row that would be returned, by using OFFSET.

```
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3 OFFSET 1;
```

Figure 4.9 shows that the row with TitleName "The Sum Also Rises" has been skipped, and the results now include a different row with the TitleName "The DateTime Machine" that has an older PublicationDate.

TitleName	PublicationDate
The Call of the While	2017-03-14 00:00:00
The Great GroupBy	2015-12-23 00:00:00
The DateTime Machine	2015-02-04 00:00:00

Figure 4.9 The most recent three TitleName and PublicationDate values after skipping the first row using OFFSET.

4.2.3 Limiting data in another RDBMS

In Chapter 1 we discussed how each RDBMS has its own variation for certain commands, and unfortunately the LIMIT clause is one of those commands.

WARNING The LIMIT clause works with many of the popular RDBMSes, including MySQL, MariaDB, PostgreSQL, and SQL Lite. However, it does not work with DB2, Oracle, or SQL Server. Those will use other commands instead of LIMIT, which are proprietary to each RDBMS.

4.3 Commenting data

Throughout this chapter we have been discussing ways to sort and skip rows in your queries. Each query has had some explanation about its purpose and considerations for executions, but if someone else were to read just the SQL we have used, would they understand why the queries were written the way they were? Probably not, which is why now would be a good time to talk about comments. *Comments* allow you to include text in your query that isn't considered for execution. Typically, this text includes some kind of note to indicate the query author's intentions, as well as their identity and the date the query was written or modified. Essentially, it can be any kind of information you want to include above and beyond the actual SQL that was written.

Why would we do this? Well, mostly because it isn't just an RDBMS that's going to read your query. People, yourself included, will read it as well. This could be another colleague who uses the script or the person who replaces you after you use your ever-expanding knowledge of SQL to secure a better job. Your comments can include something as simple as your name and the date you made your SQL script, or something as detailed as line-by-line descriptions of what each bit of SQL is intended to accomplish.

The downside of not using comments is ambiguity. Others may look at your SQL and need to spend hours trying to figure out what your intentions were. Even worse, you yourself may write some SQL and then weeks, months, or even years later look at it and be confused by what your former self wrote.

Writing descriptive and helpful comments is the mark of any well-respected SQL developer. Others will have a greater appreciation for your work, as the extra seconds or minutes you spend writing clear comments will save you and others hours of confusion in the future.

There are a few ways to write comments. First, you can comment out a particular line by using two consecutive hyphens:

```
-- This query returns three random rows
SELECT
    TitleName,
    PublicationDate
FROM title
LIMIT 3;
```

The use of two hyphens allows for the commenting of a single line of code, up to the next carriage return. This is considered an inline comment. In MySQL, you can also achieve an inline comment with the number sign (#):

```
# This query returns the 3 rows with the most recent PublicationDate
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3;
```

This type of comment isn't as common, so be aware that your RDBMS may not recognize this as a comment.

A third way to use comments is to surround your comment with /* and */, encompassing your comment between those symbols, which then allows for a multiline comment. This means you can comment out more than one line of SQL, as in the following example:

```
/* This query returns 3 TitleNames
...with the most recent PublicationDate
...excluding the single most recent TitleName */
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3 OFFSET 1;
```

TIP Because they have greater functionality, multiline comments (using /* and */) are the preferred method for use in reusable code. They can be especially useful when you want to comment out entire sections of SQL. You may want to do this to indicate a section of your SQL that doesn't execute as intended, or it's a previous version of a query that was used that you might want to reference later.

You can put multiple-line comments around single-line comments as well. For example, you might have made a one-line comment about a particular SQL statement, but later decided to comment out the entire statement using multiple-line comments, replacing it with different SQL. You could indicate this in the following way:

```
/*
# This query returns 3 random titles, but it wasn't what we needed
SELECT
    TitleName,
    PublicationDate
FROM title
LIMIT 3;*/

-- This is the updated query, now ordered by most recent PublicationDate
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3;
```

Comments can be invaluable when you write a query, save it, and then come back to it weeks, months, or even years later and review it. As someone who has been writing SQL for a few decades, I've had innumerable times where I've reviewed the comments of an old query to determine what the goal of a query was. I've also had plenty of times where I've looked at a query someone else wrote that had no comments, which led to many hours of trying to figure out what the writer had intended the query to do.

Do yourself a favor and start developing the habit of carefully commenting any SQL you write, no matter how simple. It only takes a few seconds, but as I mentioned it could save you or someone who reviews your code much more time. For this reason, all the SQL in the supplemental scripts has been commented to better help with understanding the purpose of each query.

I think we've gotten all we can out of your first query. It's time to move on to querying data that might be a bit more comparable to the kind you need to work with.

4.4 Lab

1. You need a list of all authors, but you need it to be in alphabetical order. Write a query to return the FirstName and LastName of all authors, sorted by LastName and then FirstName.
2. You need to write a query that returns all columns in the title table for only the highest-priced title. What does that query look like?
3. Suppose you have the following SQL statement that gets all the carriage returns stripped out by the application executing it, and this query ends up on a single line. What will be the result of this query?

```
-- Retrieve the book titles
SELECT TitleName
FROM title
```

4.5 Lab answers

1. The answer is below:

```
SELECT
    FirstName,
    LastName
FROM author
ORDER BY
    LastName,
    FirstName;
```

2. The answer is below:

```
SELECT
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
FROM title
ORDER By
    Price DESC
LIMIT 1;
```

Alternatively, you could use `SELECT *` instead of listing all the column names.

3. Because the entire query is now on a single line preceded by two hyphens, the entire query will execute as a comment. Although this will not result in an error, it will also not return the results the query intended. This is one of several reasons why /* and */ are advised to be used for comments in reusable code, as both the beginning and end of the commented line or lines are clearly marked.

5

Filtering on specific values

So far, we've mostly been writing queries that return an entire set of data, but as you write more purposeful SQL using larger sets of data, you will find you need only a subset of the data instead of all the rows. We did work a bit in the last chapter to reduce the number of rows returned using LIMIT and OFFSET, but those commands aren't very helpful for finding specific rows.

For instance, you may only want a report of sales for the last month, or a list of orders with pending status, or a list of customers in New Hampshire. All these scenarios have *conditions* for specific data being returned, and we apply those conditions using filtering. *Filtering* simply means taking the broader results of your data set and applying one or more conditions to restrict the data being returned, and this is primarily done using a different clause – the WHERE clause.

It's highly likely most of the SQL you write in your career will include a WHERE clause, as there are nearly an infinite number of ways you may need to find data that meets specific criteria. The WHERE clause is incredibly powerful, with so many ways to filter data that it will take us a few chapters just to review the myriad of ways it can be applied.

Let's get started!

5.1 Filtering on a single condition

The most basic methods for filtering data are relatively intuitive and easy to learn. The main variations will involve the type of data you will be querying. As you may have noticed, there are different types of data, such as names, numbers, and dates, and they each have slightly different rules for filtering. We'll look at them all in this section, starting with filtering using a condition with a numeric value.

5.1.1 Filtering on numeric values

Suppose we wanted to know the TitleName of any Titles where the Advance for the author was ten thousand dollars. Let's start by declaring a sentence.

"I would like the title name of the title where the advance is \$10,000.00."

Notice how, grammatically, we not only use the word *where* for our filtering but also place our filtering condition toward the end of the sentence. In SQL we're going to do the same thing, and we could write a query for this request like this:

```
SELECT TitleName
FROM title
WHERE Advance = 10000.00;
```

Let's take a closer look at that WHERE clause and examine the rules around it.

First, as mentioned before, the WHERE clause comes after the FROM clause, just as it naturally would in English.

Second, notice that we use an equal sign (=) in place of the word *is*. The use of the equal sign indicates equality, which means our filtering condition is looking for values equal to a specific value. In this case, the use of the equal sign clearly makes a lot of sense.

Lastly, note that we have no dollar sign or comma in 10000.00. Although we use commas to make numeric values more readable to human eyes and use dollar signs to indicate currency, this data is typically stored as a number, and the computer with your RDBMS doesn't care about a specific currency type or the readability of the numbers. In fact, using dollar signs and commas in this case would be problematic.

WARNING When you start filtering for large numeric values—for example, orders over one million dollars—it can be tempting to want to put commas in the numeric values to make the data more readable. After all, it can be easy to mistakenly type 1000000 as 100000 or 10000000. Unfortunately, including commas for numeric values will result in syntax errors for your query.

While currency types and commas can't be used with numeric values, decimals can often be added or removed without changing the results of the data. This is possible because numeric values can be equal, even if they don't have the same precision. *Precision* refers to the mathematical specificity of a value, and how precise the data is depends on how that data is stored and how you are querying it.

As an example, the number 1.00 is more precise than 1. We can increment 1.00 up to 1.01, but the next incremental value after 1 is 2. Because of this, the latter is less precise. For the purposes of querying though, even though 1.00 is more precise, it is mathematically the same as 1.

In the case of the Advance value we just used to filter, let's take a quick look at the value of the data shown with the following query, with the results shown in figure 5.1:

```
SELECT
    TitleName,
    Advance
FROM title
WHERE Advance = 10000.00;
```

TitleName	Advance
Anne of Fact Tables	10000.00

Figure 5.1 shows that only one row meets the filter criteria for an Advance value of 10000.00.

In terms of precision, the value of \$10,000.00 is more precise than \$10,000, but numerically they are the same value. Because of this, we can write a query without the decimal values used to represent cents and still get the same results that are shown in figure 5.1.

```
SELECT TitleName, Advance
FROM title
WHERE Advance = 10000;
```

TRY IT NOW

Use the previous two queries to test the WHERE clause and see how the results are the same. Also try using an even more precise value in your filter condition, such as 10000.0000. The results should all be the same.

5.1.2 Filtering on string values

So far, our queries have been filtering on numeric filter conditions, but filtering on non-numeric filter conditions is a bit different. Instead of looking for a TitleName for a specific Advance, let's reverse that and look at how we would query for the Advance of a specific TitleName, with the results shown in figure 5.2:

```
SELECT Advance
FROM title
WHERE TitleName = 'Anne of Fact Tables';
```

Advance
10000.00

Figure 5.2 The result of a query for the Advance from Title where the Title Name is Anne of Fact Tables.

Our filter condition now isn't a numeric value, but rather a group of words. To the RDBMS, this group of words is actually a group of characters known as a *string value*, and any time we filter on a string, we need to place single quotes around the value. If we don't, then it will result in a syntax error.

WARNING Not all single quotes will work. You must use the single quote on the same keyboard key as the double quotes for this to work. If you use the tick mark next to the "1/!" key on most keyboards, then you will get a syntax error. Also, if you copy and paste code from a document other than a SQL script, you may get incorrectly formatted single quotes, like the ones in figure 5.3.

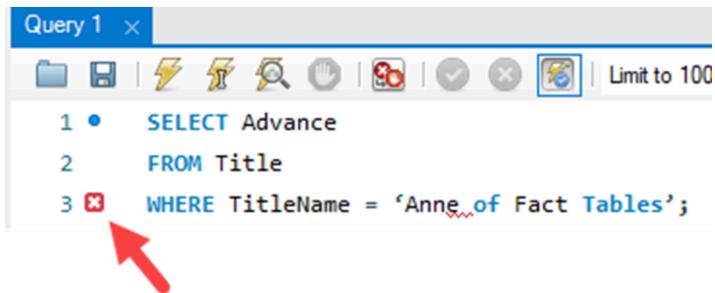


Figure 5.3 Incorrect single quotes are used, which have been copied from a Microsoft Word document. Workbench is already letting you know this with the red square with an X to the left of the line.

The value of the string used in our WHERE clause needs to be an exact match, as the slightest variation will prevent us from getting the intended results. For instance, if you forgot the letter s in Tables as in the following query, you won't get any results:

```
SELECT Advance
FROM title
WHERE TitleName = 'Anne of Fact Table';
```

NOTE Forgetting a character such as that last s may seem like a clear mistake, but there are subtle mistakes involving characters that don't seem like characters that can lead to incorrect results, such as extra spaces, tabs, or carriage returns. Although their use in your SQL is ignored by the RDBMS, the extra spaces, tabs, and carriage returns are treated as extra characters in your string values and should be avoided to match data correctly.

5.1.3 Filtering on date values

Date values have their own considerations when used for filtering, as they are used as a kind of hybrid of numeric and string values. Like string values, date and time values will also need to be enclosed in single quotes. If you wanted to find the `TitleName` with a `PublicationDate` of March 14 of 2020, you'd use the following SQL.

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate = '2020-03-14 00:00:00';
```

The default format is year-month-day and then hours:minutes:seconds. The use of single quotes here may seem obvious, as this value contains non-numeric characters, such as dashes and colons. But what if I told you the RDBMS you are using is actually storing date and time values as numeric values? It's true, as this is more efficient than storing them as a string of characters.

What this means for us and our use of SQL is that the previous rules of precision that apply to numeric values also apply to date and time values. Consider that the `00:00:00` in the filtering value represents hours:minutes:seconds—specifically, the exact second of midnight at the start of a day. If no time is provided, these zeroes are included in the default value for a date, as they are in the results of the most recent query, shown in figure 5.4:

TitleName	PublicationDate
The Call of the White	2020-03-14 00:00:00

Figure 5.4 The `TitleName` and `PublicationDate` for a title with a publication from March 14, 2020. The time values are the equivalent of midnight at the start of the day.

Now, given that we know `10,000.00` is numerically the same as `10,000`, we can conclude that `2020-03-14 00:00:00` is also the same as `2020-03-14`. Because the value of the data in the table has all zeroes for hours, minutes, and seconds, we can be confident we will get the same results as shown in figure 5.4 by writing the query without the consideration of time, like this:

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate = '2020-03-14';
```

TIP As you progress in SQL skills, it will be helpful to be aware of the kind of data included in the tables you are going to query. It's easy to see that all the values for publication date in the Title table have no precision for hours, minutes, or seconds, and that we can safely query that data without including that level of time precision. But if a single value had so much as a second of precision, it would be best to write queries that accounted for the time as well.

5.2 Filtering on multiple conditions

So far, we've filtered on only a single condition, but in real-world queries you often will need to filter on multiple conditions. For example, find a customer where the first name is Jeff and the last name is Iannucci, or the Order is number 1001 and the Item is Product X. For queries like these, the WHERE clause allows for filtering on multiple conditions using an intuitive method.

5.2.1 Filtering that requires all conditions

Suppose we want to query the Title table for Title Names where the Advance is 5000 dollars and the Royalty is 15 percent. We would verbally declare our request like this:

"I would like the Title Names from Title where the Advance is 5000 dollars and the Royalty is 15 percent."

The use of the word AND to add to our filtering criteria is intuitively included in our SQL, with the results shown in figure 5.5.

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance = 5000
    AND Royalty = 15;
```

TitleName	Advance	Royalty
Pride and Predicates	5000.00	15.00

Figure 5.5 Although multiple rows in the Title table have a Price value of \$5000, adding a second filter condition for a Royalty of 15 percent reduces the result set to one row.

In this context, the keyword AND is considered an *operator*, which means it is a keyword that performs a specific function in SQL. In the case of AND the function is joining filter conditions within the WHERE clause. This is the first of several operators we will discuss.

The use of AND allows us to add as many filter conditions as we need for a given query, even beyond the example in the previous query. Even though there is only one row in the results of the previous query, we could theoretically refine the filter further with additional criteria by adding another AND filter condition to the WHERE clause.

```
SELECT
    TitleName,
    Advance,
    Royalty,
    PublicationDate
FROM title
WHERE Advance = 5000
    AND Royalty = 15
    AND PublicationDate = '2015-04-30';
```

TRY IT NOW

Use the previous two queries to test the WHERE clause and see how the results change with each filter. Try filtering first with a condition where the Royalty is 12 percent, then add another filter condition where Advance is 6000 dollars. The rows in the result set should reduce from two rows in your first query to one row in the second.

Although there is no specific limit to how many filter conditions you can include in your WHERE clause, understand that in order for rows to be included in your result set, they must meet every one of the filter conditions. Failure to meet any single condition will exclude the rows from the results.

5.2.2 Filtering that requires any one of many conditions

While the AND operator allows us to filter on multiple conditions that all rows must meet, sometimes we will want to apply multiple filter conditions but want results that simply meet *one or more* of the conditions.

For example, what if we wanted to find any book that either has an Advance of 5000 dollars or a Royalty of 15 percent? We could verbally declare the request like this:

“I would like the Title Names from Title where the Advance is 5000 dollars or the Royalty is 15 percent.”

The use of the word “or” has replaced “and” in our sentence, and it will do the same in our SQL:

```

SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance = 5000
    OR Royalty = 15;

```

TitleName	Advance	Royalty
Pride and Predicates	5000.00	15.00
Catcher in the Try	5000.00	10.00
Anne of Fact Tables	10000.00	15.00
The DateTime Machine	5500.00	15.00
The Call of the While	2500.00	15.00
The Sum Also Rises	5000.00	12.00

Figure 5.6 The results for any rows in the Title table that have either an Advance of \$5000 or a Royalty of 15 percent.

If this query is executed, note in figure 5.6 that we will have very different results than the previous query, where the AND operator was used instead of the OR operator. We now have six rows returned instead of one because rows in the results only need to meet either condition to be included – not both.

The OR operator can be used in the same way as the AND operator, in that we can add as many filter conditions as our query requires. Note that for each AND operator added, a result set could potentially get smaller and smaller as the conditions become more restrictive, while each OR operator could result in larger and larger result sets, as the results become more inclusive.

We can increase the result set from six rows to seven by adding another OR operator for Price like this, because now any given row included in the results in figure 5.7 needs to meet at least one of three filter conditions to be included:

```

SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance = 5000
    OR Royalty = 15
    OR Price = 9.95;

```

TitleName	Advance	Royalty	Price
Pride and Predicates	5000.00	15.00	9.95
The Join Luck Club	6000.00	12.00	9.95
Catcher in the Try	5000.00	10.00	8.95
Anne of Fact Tables	10000.00	15.00	12.95
The DateTime Machine	5500.00	15.00	7.95
The Call of the While	2500.00	15.00	8.95
The Sum Also Rises	5000.00	12.00	7.95

Figure 5.7 The seven rows that can match any one of the three conditions of an Advance of \$5000, a Royalty of 15 percent, or a Price of \$9.95.

We can go on and on adding filter conditions with multiple OR statements, and each time we will get as many or more rows, as the filter conditions become more and more inclusive.

5.2.3 Controlling the order of multiple filters

There will be situations where we need to filter in a way that requires using both AND and OR in the WHERE clause, but we need to be very careful about how we do this.

Suppose we need to find a list of Title Names where the Price of the title is \$9.95 and either the Publication Date is February 6, 2016, or the Advance is 6000 dollars. To find this data, we might try writing a query like this:

```
SELECT
    TitleName,
    Price,
    PublicationDate,
    Advance
FROM title
WHERE Price = 9.95
    AND PublicationDate = '2016-02-06'
    OR Advance = 6000;
```

This looks correct, but if we execute the query, we will find the results don't match the logic. The reason is the RDBMS will read this query differently than we intended, as shown by the results in figure 5.8.

TitleName	Price	PublicationDate	Advance
Pride and Predicates	9.95	2015-04-30 00:00:00	5000.00
The Join Luck Club	9.95	2016-02-06 00:00:00	6000.00
Catcher in the Try	8.95	2017-04-03 00:00:00	5000.00
The Sum Also Rises	7.95	2021-11-12 00:00:00	5000.00

Figure 5.8 The rows returned from the query do not match our intended filter conditions for Title Names with a Price of \$9.95 and either a Publication Date of February 6, 2016, or an Advance of \$5000.00.

This is because the RDBMS will prioritize AND conditions over OR conditions, regardless of our intentions. Let's list the logic of our intended filtering conditions, either of which needed to be met.

1. A Price of \$9.95 and a Publication Date of February 6th, 2016
2. A Price of \$9.95 and an Advance of 5000 dollars

However, since the RDBMS places a higher priority on AND conditions over OR conditions, it determines the written filtering conditions differently from what was intended. To the RDBMS, our SQL is requesting rows that meet either of these conditions.

1. A Price of \$9.95 and a Publication Date of February 6th, 2016
2. An Advance of \$5000.00

The results in figure 5.8 show the one title that met the first condition (*The Join Luck Club*) and three others that met the second condition.

Getting this logic correct can often be one of the most confusing issues for SQL beginners, but ironically the solution is very simple. All you need to do is use parentheses to explicitly prioritize your logic over the default SQL logic. Anything inside the parentheses will be evaluated before anything outside the parentheses.

To get the results we intended, we would use parentheses in the previous query like this:

```
SELECT
    TitleName,
    Price,
    PublicationDate,
    Advance
FROM title
WHERE Price = 9.95
    AND (PublicationDate = '2016-02-06'
        OR Advance = 5000);
```

This query will now evaluate the values as intended, evaluating the OR condition before the AND condition. Executing this query returns a result set like the one shown in figure 5.9.

TitleName	Price	PublicationDate	Advance
Pride and Predicates	9.95	2015-04-30 00:00:00	5000.00
The Join Luck Club	9.95	2016-02-06 00:00:00	6000.00

Figure 5.9 With the parenthetical notation, the query now returns data that has a Price of \$9.95 and either a Publication Date of February 6th, 2016, or an Advance of \$5000.00.

TIP Whenever you write any SQL using both AND and OR operators in your WHERE clause, you should always use parentheses to explicitly control the evaluation. This not only helps the RDBMS figure out your intentions but also reduces the guesswork for anyone else who will be evaluating your code.

5.2.4 Filtering and using ORDER BY

As we've gone through this chapter, you might have found yourself wondering what happened to the ORDER BY clause we used in the previous chapter, and how it fits in with the WHERE clause we've been using in this chapter. When using both WHERE and ORDER BY clauses in your SQL, the ORDER BY clause needs to be written *after* the WHERE clause.

We can use a query from earlier in the chapter that returned four rows. Figure 5.7 shows the rows returned in no particular order, but if we wanted the results sorted by TitleName, we could easily add an ORDER BY clause like this, with the results shown in figure 5.10:

```
SELECT
    TitleName,
    Price,
    PublicationDate,
    Advance
FROM title
WHERE Price = 9.95
    AND PublicationDate = '2016-02-06'
    OR Advance = 6000
ORDER BY TitleName;
```

TitleName	Price	PublicationDate	Advance
Catcher in the Try	8.95	2017-04-03 00:00:00	5000.00
Pride and Predicates	9.95	2015-04-30 00:00:00	5000.00
The Join Luck Club	9.95	2016-02-06 00:00:00	6000.00
The Sum Also Rises	7.95	2021-11-12 00:00:00	5000.00

Figure 5.10 The four rows that match any one of the three conditions of a Price of \$9.95, a Publication Date of April 30, 2015, or an Advance of \$5000.00, and the results are sorted alphabetically by TitleName.

We've covered a lot of examples of basic filtering today. It's time to use your new skills!

5.3 Lab

1. If we don't place single quotes around a non-numeric string value in our filter condition, we know we will get a syntax error. Try placing single quotes around a numeric value like Price in a filter condition and executing a query. What happens?
2. Why does the following query return no results?

```
SELECT
    TitleName,
    Price
FROM Title
WHERE TitleName = 'Anne of Fact Tables ';
```

3. What will be the result of the following query?

```
SELECT TitleName
FROM Title
ORDER BY TitleName ASC
WHERE Price = 9.95;
```

4. Write a query using the author table that returns rows with either a PaymentMethod of Check and a FirstName of Jorge, or a PaymentMethod of Check and a LastName of Miller. Include the columns FirstName, LastName, and PaymentMethod in your result set.

5.4 Lab answers

1. The query executes as expected, although behind the scenes the RDBMS has made the data match. This means it had to convert either the value in quotes to a number, or all the values in the Price column to a string, which could negatively impact the duration of the query, if we were using a larger set of data. For this reason, you should avoid putting single quotes around numeric values.
2. The query returns no values because there is an extra space after the word "Tables" in the WHERE clause. For a string used in a filter to be correct, it needs to be exactly as the values are in the table, including unseen characters, such as spaces, tabs, and carriage returns.
3. The query will result in a syntax error, as the ORDER BY cannot come before the WHERE clause.

4. This is similar to the query used in section 5.2.3. Using AND and OR and parentheses, your query should return two rows and should look something like this:

```
SELECT
    FirstName,
    LastName,
    PaymentMethod
FROM author
WHERE PaymentMethod = 'Check'
AND (FirstName = 'Jorge'
     OR LastName = 'Miller');
```

6

Filtering with multiple values, ranges, and exclusions

As shown in the last chapter, the WHERE clause offers many useful options for filtering results based on specific conditions. We looked at several examples of how to filter on a single value using the AND and OR operators, but in this chapter, we're going to expand that concept to search on multiple values. This includes a list of specific values, or ranges of unspecified values.

These are all examples of *positive searches*, where we want to match values that we want to see in the results of our queries. Because there will often be times when we want to do the opposite and see all the values except some specific filter conditions, we're going to see how to negate any of the conditions we've covered.

Let's start by looking at a new operator for the WHERE clause.

6.1 Filtering on specific values

We previously looked at a basic search, such as finding the Title Names for titles that had a certain Price. For example, if we wanted to query titles that had a Price of \$10.95, we would write our SQL like this:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price = 10.95;
```

But what if we wanted to find the titles with a Price of either \$10.95 or \$12.95? We now know we can write SQL to do this using the OR operator, so we could write a SQL query like this, resulting in the output shown in figure 6.1:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price = 10.95
    OR Price = 12.95;
```

TitleName	Price
Anne of Fact Tables	12.95
The Great GroupBy	10.95

Figure 6.1 The results of a query with filter conditions for a Price of \$10.95 or \$12.95.

This will give us the results we want for our two filter condition values, but this method could lead to some *wordy* SQL if we have a list of conditions that grows much longer. We don't want to be using an OR operator if we have three, ten, or even more filter condition values for a single column.

To resolve this, SQL has the IN operator, which allows us to consolidate our filter conditions into a single operator. Using the IN operator has three requirements:

- The list of values must be comma delimited.
- The list of values must be enclosed in parentheses.
- The use of single quotes is the same as in any other filter condition.

As an example, we can rewrite the preceding query with an IN operator, like this:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price IN (10.95, 12.95);
```

Executing this query will yield the same results as shown in figure 6.1, but in a more compact form of SQL. As mentioned, the use of single quotes would be the same as when we used them in the previous queries that filtered for filter conditions with string or date values.

TRY IT NOW

Using the IN operator, execute a query looking for Title Name and Price from Titles where Price is \$7.95, \$8.95, or \$9.95.

If we wanted to search for titles with a specific Publication Date, we need to use single quotes with our filter conditions, since we are using data values instead of numeric values. We would write our SQL like this, with the results shown in figure 6.2:

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate IN ('2015-04-30', '2016-02-06');
```

TitleName	PublicationDate
Pride and Predicates	2015-04-30 00:00:00
The Join Luck Club	2016-02-06 00:00:00

Figure 6.2 The results for titles with a Publication Date of April 30th, 2015, or February 6th, 2016.

NOTE As far as the RDBMS is concerned, the order of values used with the IN operator is irrelevant. As with any query that doesn't involve an ORDER BY clause, the rows included in the result set are not guaranteed to be returned in any particular order. That said, as has been noted several times in this book, you should follow best practices to make your SQL more readable by humans whenever possible. For this reason, it is best to list values in the filter conditions of the IN operator in numerical, alphabetical, or chronological order.

6.2 Filtering on a range of values

Filtering on a list of specific values with the IN operator is useful if you know all the specific values you want to match, but not when you don't know all the specific values. Often you will need to find values with a range, like any values higher than a certain amount or older than a certain date. There are comparison operators we can use in SQL to determine these.

6.2.1 Filtering on an open-ended range

Assuming that you have worked with even basic math, then you are familiar with the *less than* (<) and *greater than* (>) signs. (On a typical keyboard they share the same keys as the comma and the period, respectively.) We can use these two signs in place of the equal sign (=) to find an open-ended range of values.

In a query earlier in this chapter, we looked for any title that had a Price of \$10.95 or \$12.95. In our title table, these are the titles with the highest price. We can find these same two titles by writing a query using the greater than sign. Let's also include an ORDER BY clause to see the Price values in order, recalling that the default is to order the results by ascending values, and see the output in figure 6.3:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price > 9.95
ORDER BY Price ASC;
```

TitleName	Price
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Figure 6.3 The titles that have a Price greater than \$9.95, ordered by Price ascending.

We can use the less than sign to do the opposite and find any titles that have a price of less than \$9.95. For this query we will sort by descending Price values to show the Prices closest to \$9.95 at the top of the results shown in figure 6.4:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price < 9.95
ORDER BY Price DESC;
```

TitleName	Price
Catcher in the Try	8.95
The Call of the White	8.95
The DateTime Machine	7.95
The Sum Also Rises	7.95

Figure 6.4 The titles that have a Price less than \$9.95, ordered by Price descending.

You probably noticed that neither of these result sets included any titles that matched the price of \$9.95. There will certainly be times when you want to query a range of values that includes values that match the filter conditions, so SQL also includes the comparison operators of *less than or equal to* (\leq) and *greater than or equal to* (\geq).

Using one of these operators, we can now query any title with a price greater than or equal to a filter condition of \$9.95, with the results shown in figure 6.5:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price >= 9.95
ORDER BY Price ASC;
```

TitleName	Price
Pride and Predicates	9.95
The Join Luck Club	9.95
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Figure 6.5 The titles that have a Price greater than or equal to \$9.95, ordered by Price ascending.

NOTE You can obviously see how we can search for values greater than or less than a particular numeric or date value. Although the use cases are less common, you can also use these operators with string values. The values are typically returned with respect to precedence in the alphabet; however, be careful when using these operators with string values, as how the numeric, symbol, and case values are filtered is based on the collation settings in your RDBMS. *Collation* determines the rules for sorting, based on character qualities such as the character set, letter case, and accent usage; therefore, different collations will return characters in a different order.

6.2.2 Filtering a defined range

The operators in the preceding section were *open-ended*: that is, they could potentially include an infinite number of values greater than or less than a value. If we want to find values greater than some value but also lesser than another value, we have a couple of options.

The first one you may have already thought of, which is to use both `>` and `<` in the WHERE clause. If we are trying to find titles with a price between \$8.95 and \$10.95, then we can write a SQL query like the following. Again, we will use an ORDER BY in the query to organize the results so that we can see the maximum and minimum values returned by our filter conditions in figure 6.6:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price > 8.95
    AND Price < 10.95
ORDER BY Price ASC;
```

TitleName	Price
Pride and Predicates	9.95
The Join Luck Club	9.95

Figure 6.6 The titles with Price values between the search conditions of \$8.95 and \$10.95.

We can see that the only price value that matches the range of our filter conditions is \$9.95. If we wanted to change the filter conditions to *include* the values we've specified, then we need to use `>=` and `<=` in the WHERE clause like this:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price >= 8.95
    AND Price <= 10.95
ORDER BY Price ASC;
```

TitleName	Price
Catcher in the Try	8.95
The Call of the White	8.95
Pride and Predicates	9.95
The Join Luck Club	9.95
The Great GroupBy	10.95

Figure 6.7 The titles with Price values between the filter conditions of \$8.95 and \$10.95, when the values used in the filter condition are included.

As figure 6.7 shows, we now have titles included in the results set that match the Price values used in the filter conditions.

There is another way to search a range of values like this. Instead of using `>=` and `<=`, we can use the BETWEEN operator to perform the same function. The rules for using BETWEEN are:

1. The column being searched is only mentioned once.
2. There can be only two filter conditions supplied.
3. The first value will represent the low end of the range, and the second value will represent the high end of the range.
4. The filter conditions must be separated by the word AND.
5. Any values matching either condition will be included in the results.

We can write the previous query using the BETWEEN operator like this:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price BETWEEN 8.95 AND 10.95
ORDER BY Price ASC;
```

Even though we used one less line of SQL, the results of this query should be identical to the results of those shown in figure 6.7.

WARNING It should be mentioned again that when using BETWEEN, the first value represents the low end of the range and the second represents the high end of the range. If you use BETWEEN and the first value of the filter conditions of the range is higher than the second value, then your query will return no results. For example, having a query with SQL that says WHERE Price BETWEEN 10.95 and 8.95 will not return any rows, regardless of other filter conditions. Logically, there are no values that are both higher than 10.95 and lower than 8.95.

6.3 Negating filter conditions

So far, we have used the WHERE clause to specify filter conditions that match specific values or ranges of data. These are known as *inclusive conditions*, since the results include rows with values that match the values we have used in our filter conditions. The opposite of these are known as *exclusive conditions*, where we want all values *except* the ones specified in the filter conditions.

In a way, this is a bit like how we used OFFSET in Chapter 4 to exclude a specific number of rows, but with much more specificity about what we are excluding. Let's look at the ways we can use exclusive filter conditions in SQL.

6.3.1 Negating a specific value

In Chapter 5, we learned to filter on specific values using the equal sign (=). Mathematically, the way we can represent the opposite is by using the *not equal sign* (\neq), which combines the less than and greater than signs together.

If we wanted to list all titles that did not have a Price of 7.95, we could use not equals like this, with the ordered results shown in figure 6.8:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price <> 7.95
ORDER BY Price ASC;
```

TitleName	Price
Catcher in the Try	8.95
The Call of the White	8.95
Pride and Predicates	9.95
The Join Luck Club	9.95
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Figure 6.8 All titles, excluding the two that have a price of 7.95.

Even though we've used \neq for mathematical comparisons, we can use it with date and string values as well. For instance, if we wanted all titles except any published on February 6, 2016, then we can use single quotes with \neq to get the desired results shown in figure 6.9:

```

SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate <> '2016-02-06'
ORDER BY PublicationDate ASC;

```

TitleName	PublicationDate
The DateTime Machine	2019-02-04 00:00:00
The Sum Also Rises	2021-11-12 00:00:00
Catcher in the Try	2017-04-03 00:00:00
The Call of the White	2020-03-14 00:00:00
Pride and Predicates	2015-04-30 00:00:00
The Great GroupBy	2019-12-23 00:00:00
Anne of Fact Tables	2018-01-12 00:00:00

Figure 6.9 All titles, excluding the one (“The Join Luck Club”) that has a Publication Date of February 6th, 2016

NOTE The RDBMS you use may also offer the option to use != instead of <>. They both perform the same function of negating a single condition, but it is advisable to develop the habit of using <> in your SQL, as != is not supported in every RDBMS.

6.3.2 Negating any filter condition

While <> offers the ability to exclude a single value in a filter condition, there is one operator that excludes an entire filter condition. This is the NOT operator, and it turns any inclusive filter condition into an exclusive condition.

For example, in the preceding query, we used <> to exclude any title that has a Publication Date of February 6, 2016. We can use the opposite of <>, which would be =, with the NOT operator to achieve the same results, as shown in figure 5.9 with the results ordered by PublicationDate.

```

SELECT
    TitleName,
    PublicationDate
FROM title
WHERE NOT PublicationDate = '2016-02-06'
ORDER BY PublicationDate ASC;

```

In this instance, the NOT operator immediately followed the word WHERE. You can use the NOT operator after the start of any filter condition to negate it, which means it could also be used immediately after conditions that begin with AND or OR operators.

TRY IT NOW

Execute the two previous queries to see how `<>` and NOT can be used to exclude specific values.

As mentioned in a previous Note paragraph, you probably wouldn't use the negative NOT operator with `>` or `<`, since you can logically use their positive opposites of `<=` and `>=` respectively to get the same results with simpler syntax. However, one common way NOT is used is with the IN operator mentioned earlier in this chapter.

Recall how we used a single condition using the IN operator to query all titles with a Price of either \$10.95 or \$12.95 to replace two conditions using the OR operator. We can use NOT to negate this filter condition with the IN operator, returning all titles excluding those matching the conditions of a Price of \$10.95 or \$12.95, with the results sorted by Price in figure 6.10:

```
SELECT
    TitleName,
    Price
FROM title
WHERE NOT Price IN (10.95, 12.95)
ORDER BY Price ASC;
```

TitleName	Price
The DateTime Machine	7.95
The Sum Also Rises	7.95
Catcher in the Try	8.95
The Call of the White	8.95
Pride and Predicates	9.95
The Join Luck Club	9.95

Figure 6.10 All titles, excluding those with a Price of either \$10.95 or \$12.95.

You may have noticed the condition WHERE NOT Price IN in the previous query sounds a bit clumsy to the English speaker. There is a better, and more often used, way in SQL to get the same results. NOT IN is actually a separate operator, so we can also place NOT before the IN in our filter condition, like this:

```
SELECT TitleName, Price
FROM title
WHERE Price NOT IN (10.95, 12.95)
ORDER BY Price ASC;
```

This returns the same results as shown in figure 6.10. In cases where we want to use an exclusionary list of values, such as in the last example, using NOT IN is the more commonly used operator.

6.4 Combining types of filter conditions

We've shown several new ways to filter your data with inclusive and exclusive filter conditions. One last point to make is that you can and will be combining both kinds of filter conditions in your queries.

For example, we could write a query where we want to find any title that has an Advance value > 5000 but a Royalty not equal to 12%, with the results shown in figure 6.11.

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance > 5000
    AND Royalty <> 12;
```

TitleName	Advance	Royalty
Anne of Fact Tables	10000.00	15.00
The DateTime Machine	5500.00	15.00

Figure 6.11 The titles that have an Advance greater than \$5000 with a Royalty that is not 12%.

With all the operators discussed in this chapter and the previous one, you can begin to write some relatively complex filter conditions. If you wanted to include the previous results (any title that has an Advance value > 5000 but a Royalty not equal to 12%) and also any titles published after January 1 of 2020, you can easily do that with the SQL covered so far for inclusive and exclusive queries, and by controlling the logic with parentheses. If you tried this, your SQL query might look something like the following, with the results shown in figure 6.12:

```

SELECT
    TitleName,
    Advance,
    Royalty,
    PublicationDate
FROM title
WHERE (Advance > 5000
    AND Royalty <> 12)
    OR (PublicationDate > '2020-01-01');

```

TitleName	Advance	Royalty	PublicationDate
Anne of Fact Tables	10000.00	15.00	2018-01-12 00:00:00
The DateTime Machine	5500.00	15.00	2019-02-04 00:00:00
The Call of the White	2500.00	15.00	2020-03-14 00:00:00
The Sum Also Rises	5000.00	12.00	2021-11-12 00:00:00

Figure 6.12 The titles with an advance of \$5000 that don't have a royalty of 12%, or any title published after January 1 of 2020.

TIP Although exclusive filter conditions can be used to achieve the same results as inclusive filter conditions, it is preferable to use inclusive filter conditions when possible. Inclusive conditions are not only often easier for others who read your SQL to comprehend but are also, for the most part, processed more efficiently by the RDBMS. The exception would be if you had only one or a handful of exclusions to filter, in which case it is likely better to use exclusive filters instead of an inclusive filter with a vast number of inclusive conditions or values.

We're only a few chapters into the book, but you already have the capability to go and search data in meaningful and accurate ways. We've mostly used filter conditions with numeric and date values so far, but in the next chapter we'll discuss another series of tools we can use in the WHERE clause to perform advanced searches for data in string values.

6.5 Review of comparison operators

In this chapter, we have covered more than a dozen comparison operators that can be used for filtering in the WHERE clause. You may have been taking lots of notes, but in case you didn't, table 6.1 presents a list of what we have used.

Table 6.1 Review of WHERE clause comparison operators

Operator	Description
=	Equality
<>	Inequality
!=	Inequality*
<	Less than
>	Greater than
!<	Not less than*
!>	Not greater than*
<=	Less than or equal to
>=	Greater than or equal to
BETWEEN	Between two values, including those values
IN	Equality to a list of multiple values
NOT IN	Inequality to a list of multiple values
NOT	Inequality to stated condition

*May not be supported by every RDBMS

6.6 Lab

You have only one lab assignment today, but it's a larger challenge to creatively use what you've learned so far. In the last two chapters, we have covered quite a few ways to include and exclude data, so consider all that you've learned about using the WHERE clause for filtering.

For this exercise, try to think of as many different ways as possible to use the WHERE clause to return the TitleName and Price for all rows in the title table that do not have a price of \$9.95. The results of each query should include only the rows shown in figure 6.13, ordered by Price.

TitleName	Price
The DateTime Machine	7.95
The Sum Also Rises	7.95
Catcher in the Try	8.95
The Call of the White	8.95
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Figure 6.13 The titles with a Price that is not \$9.95, ordered by Price ascending.

6.7 Lab answers

These are some of the many ways you can write the WHERE clause to exclude titles with a price of \$9.95:

- WHERE Price <> 9.95
- WHERE NOT Price = 9.95
- WHERE Price < 9.95 OR Price > 9.95
- WHERE PRICE NOT IN (9.95)
- WHERE Price IN (7.95, 8.95, 10.95, 12.95)
- WHERE Price BETWEEN 7.95 AND 8.95 OR Price BETWEEN 10.95 and 12.95
- WHERE NOT Price BETWEEN 9.95 and 9.95

That last one might seem a bit unexpected, as it effectively negates a range of values that are only a single value. It isn't a method you would often use, as a filter condition on a single value is typically done using an = in the WHERE clause. It's shown here only to demonstrate that a range with the same high and low end can be executed.

7

Filtering with wildcards and null values

The last few chapters have been filled with different ways to filter the data returned by your queries using numerous comparison operators. We've worked with many methods for filtering on one or more values of equality or inequality using known values or ranges of values. Let's use one more chapter to examine some of the more interesting ways to search for less specific data.

In this chapter, we're going to look at how to filter data when you don't know the exact values to be searched. Instead of searching for specific values, we're going to search for *patterns of values*. This can be incredibly useful when you want to look for a list of products that have specific text like *tomato* or *cable* in the name, or when you want a list of all customers whose last name starts with the letter A.

We're also going to look at the trickiest value to search on: null. Null values are commonly misunderstood, and as such, they can often lead to incorrect query results. We're going to examine what a null value is (and isn't), and how to properly query columns that have null values.

7.1 Filtering with wildcards

In the previous chapter, you learned how to search ranges of numeric or date values. Even though you might not know all the specific values we want from a range, you now know how to query the correct results using operators such as `>`, `<`, and `BETWEEN`.

Interestingly, we can use those same operators to search string values. For example, if we wanted to find all the first and last names of authors with a last name that starts with an S then we could write using `>=` and `<` to get the result shown in figure 7.1:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName >= 'S'
    AND LastName < 'T';
```

FirstName	LastName
Jen	Strong
Gail	Shawn

Figure 7.1 The results of searching the author table for a range of last names that start with S

NOTE From here on out, we won't sort results unless it is necessary. As stated earlier, sorting data with ORDER BY increases the work required to process any query, so it should be avoided if possible. Just remember that without ORDER BY, it is always possible that you may get the same results in a different order for any given query we execute.

This method of searching for string values in a range likely works most of the time, but not always. As briefly noted in the last chapter, depending on the collation settings, the character case (upper or lower), and the characters used (such as letters with tildes or umlauts), you may not get consistent results using this method for filtering on string values.

Also, it just looks weird to write a query this way, and we certainly wouldn't verbally declare that this is how we want to filter results. We want a list of last names that start with S and not a list of names between S and T. This is where a wildcard makes more sense.

A *wildcard* is a special character that can be substituted for some number of characters in a string. The use of a wildcard allows us to search for specific patterns of values instead of being restricted to a range, as in the preceding query.

7.1.1 Filtering with the percent sign (%)

The first wildcard we're going to use is %, the percent sign. When used as a wildcard, the percent sign matches any string, including an empty string with no characters. Here's how we would use it to find the names of authors with a last name that starts with "S".

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE 'S%';
```

Notice that we are now using a new operator, LIKE. LIKE is the operator we will always use when searching with a wildcard, because in the SQL language it indicates we are searching for a pattern and not for precise conditional values. If you tried to use some other conditional operator such as = or > in this query instead of LIKE, you would get no results.

TRY IT NOW

Execute the previous query, then try using the = operator instead of LIKE.

Even though we know how the query ends, let's take a moment to compare this with how we might verbally declare this query:

"I would like the first name and last name from the author table where the last names start with S."

I can assure you there is no STARTS WITH operator. And although it might be useful for our query, it would not be very flexible. We would also need hypothetical operators for other queries, like ENDS WITH and maybe even HAS IN THE MIDDLE. These would be excessively wordy, if not a bit ridiculous.

In SQL, the % operator is not only shorter but also has the same functionality as all those other hypothetical operators. The easiest way to remember how to use it is to think of the % wildcard as the word *something*. The *something* pattern we are looking for could be zero characters or 100. Here's a verbal way to say what we're doing with the query:

"I would like the first name and last name from the author table where the last names are like S and then something."

This is fairly close to what our new query looks like, and the results will be the same as those in figure 7.1. As just mentioned, the % wildcard can be used anywhere in a string of characters, which means if we want to search for all last names that end in an "N" we could verbally declare a query like this:

"I would like the first name and last name from the author table where the last names are like something and then N."

As you can imagine, our query would be very similar to the verbal declaration. Here it is, along with the results in figure 7.2:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE '%N';
```

FirstName	LastName
Robert	Davidson
Gail	Shawn
Andy	Melkin
Deepthi	Mahadevan

Figure 7.2 Authors who have a last name that ends with N

If we wanted to refine this search to include only the last names that not only end with "N" but also start with "M", like the results shown in figure 7.3, we can certainly do that as well:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE 'M%N';
```

FirstName	LastName
Andy	Melkin
Deepthi	Mahadevan

Figure 7.3 Authors who have a last name that starts with M and ends with N

We can also use the % operator both before and after a character or string of characters to find a pattern in the middle of our data. Here is an example of searching for authors with the string of "de" anywhere in their last name, with the results shown in figure 7.4:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE '%DE%';
```

FirstName	LastName
Buck	Fernandez
Deepthi	Mahadevan

Figure 7.4 Authors who have a last name that contains "DE"

This can be useful when searching a column of comments or other freely entered text. For instance, if you wanted to find any comments that used the word “good” you would search where the column values were LIKE ‘%good%’. As noted before, most RDBMSs will not be case sensitive, so you should be able to return values including “Good” or “GOOD.” Then again, you might also get string values like “not very good” or “goodbye” in your results, since they also match the pattern.

WARNING Although the LIKE operator is not case sensitive in the default collations of MySQL, Microsoft SQL Server, and SQLite databases, it can be case sensitive in the default collations of PostgreSQL and Oracle databases.

As helpful as the % wildcard can be for finding patterns of characters, it lacks precision. If we want to search values at a particular position, there is a different wildcard we can use.

7.1.2 Filtering with an underscore (_)

Where the % wildcard matches any string of characters (including zero characters), the _ wildcard, an *underscore*, will only look for any single character. What’s more, we can combine _ with % in our search patterns if necessary.

WARNING The underscore wildcard is not supported in DB2.

For example, if we wanted to find the first and last names of any author with a first name that starts with *R* and has *b* as the third letter, as shown in figure 7.5, we can use _ and % to find those.

```
SELECT
    FirstName,
    LastName
FROM author
WHERE FirstName LIKE 'R_b%';
```

FirstName	LastName
Robert	Davidson
Rebecca	Miller

Figure 7.5 Authors who have a first name that starts with *R* and with *b* as the third letter

Although the _ wildcard is generally less often used than the % wildcard, you can still face scenarios of searching for patterns at one specific position—for instance, if you needed to find items with a color value of gray, which could be spelled as “gray” or “grey.” To find all matching values, you could search WHERE color LIKE “gr_y”.

You can also search for values or locations that might have a variance in the first few characters using the _ wildcard. Here is an example of finding any author with a first name that has a *u* as the third character, with the results shown in figure 7.6:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE FirstName LIKE '__u%'
ORDER BY FirstName ASC;
```

FirstName	LastName
Doug	Li
Paul	Tripp

Figure 7.6 The results of authors who have a first name that has a *u* as the third letter

Other wildcards beyond % and _ that are supported within each RDBMS, but because they vary in their inclusion, we won’t discuss them in this book. That being said, if you are using a particular RDBMS, then I would encourage you to look into what other wildcards may be supported, to further enhance your ability to search for patterns of values.

Now let’s move on to, well, nothing.

7.2 Filtering with null values

Sometimes database designers create columns that require values for every column, but there are other times when a column may allow *null values*. This means any given row may or may not have the property represented by the column, and any row that does not will show NULL for that column.

As noted earlier, null values are some of the most misunderstood concepts in databases. Put simply, NULL values are literally nothing: They represent the absence of data. This seems simple to understand, but because null values are not values like 30 or Arizona or May 12th of 2012, they need to be considered differently from other values when querying data.

Let's look at an example by reviewing all the columns in the author table. Executing the following query will return all the columns for all 11 rows in the author table. One of the first things you might notice is the MiddleName column, which has quite a few values that say NULL. Not everyone has a middle name, so the absence of a middle name for any author is represented by NULL. The Workbench tries to bring this to your attention by making null values appear different from other values we've seen so far, in that they are shown with white text in a smaller font size and a dark background color as in figure 7.7:

```
SELECT *
FROM author;
```

AuthorID	FirstName	MiddleName	LastName	PaymentMethod
1	Paul	K	Tripp	Cash
2	Doug	NULL	Li	Check
3	Jen	NULL	Strong	Check
4	Jorge	Armando	Guerra	Check
5	Robert	Grant	Davidson	Check
6	Gail	Anne	Shawn	Check
7	Rebecca	NULL	Miller	Check
8	Andy	NULL	Melkin	Direct Deposit
9	Buck	NULL	Fernandez	Cash
10	Chris	NULL	Walenski	Direct Deposit
11	Deepthi	NULL	Mahadevan	Direct Deposit

Figure 7.7 The results of all columns in the author table, including null values in the MiddleName column

7.2.1 How not to search for null values

As stated earlier, a null value represents the absence of a value, which makes any columns containing null values tricky to query. To avoid some common pitfalls, let's first talk about how not to query for NULL values. If we want to find the rows in the author table that contain NULL values for MiddleName, none of the next three examples will work:

```
/* This doesn't work because null values are not blank strings. */
SELECT *
FROM author
WHERE MiddleName = '';
```

This query won't return null values because the query is actually searching for a character string with a length of zero, also known as an *empty* string. I know—that's confusing because when I mention "absence of a value" and "empty string" it seems like I'm saying the same thing in different ways. But an empty string is definitely different from a null value, as an empty string is still a string. By that, I mean that underneath the covers, your RDBMS is still using bytes to indicate a value for the empty string, and as such it can be considered for queries that are filtering with many comparison operators, including a wildcard search. Null values use no bytes and will not be considered for filtering with comparison operators or wildcards.

Here's another common but incorrect way to search for null values.

```
/* This doesn't work because null values are not the word null. */
SELECT *
FROM author
WHERE MiddleName = 'NULL';
```

This doesn't work because the search condition is for the word "NULL" and not a null value. This also won't return any rows unless your data is populated with a string of the four characters that make the word "NULL". That may seem unlikely, but sometimes database developers don't understand how to work with null values, so they use the word "NULL" to represent null values. Since the word "NULL" is a string of characters, this can create all sorts of headaches for your queries. Please don't ever do this.

Here is one last incorrect way to search for null values.

```
/* This doesn't work because no value ever equals null. */
SELECT *
FROM author
WHERE MiddleName = NULL;
```

This seems like it should work, but = is looking for equality, and you can't have equality matches of nothing. At the most basic level, any of the operators including = are evaluating for search conditions that are either true or not true. Because a null value is nothing, it won't ever equal anything in a search condition, so it will never be evaluated as being true.

TRY IT NOW

Execute any or all of the three previous queries, and see that they don't return any matching rows.

7.2.2 How to correctly search for null values

To correctly search for null values, let's state another verbal example in English for what we're trying to do. If we wanted the full names of any author who has a null value for a middle name, we could say the following:

"I would like the first, middle, and last name of authors where the middle name is null."

To turn previous verbal declarations like this into SQL queries, we have replaced "is" with the = operator. Since we're dealing with a filter condition involving null values, which don't work with comparison operators, we can instead use a new operator that is literally the last two words of the verbal declaration: IS NULL. Here's what our query will look like, with the results shown in figure 7.8:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName IS NULL;
```

FirstName	MiddleName	LastName
Doug	NULL	Li
Jen	NULL	Strong
Rebecca	NULL	Miller
Andy	NULL	Melkin
Buck	NULL	Fernandez
Chris	NULL	Walenski
Deepthi	NULL	Mahadevan

Figure 7.8 The results of first, middle, and last names in the author table for authors with no middle name

Pay close attention to null values and the IS NULL operator because null values can be even more problematic. Refer to figure 7.7 and notice the first of the 11 rows in the author table has a middle name value of K. Suppose you want to query all rows except that one, with an exclusion query like we learned about in the last chapter. You could do this with the following query, with the results shown in figure 7.9:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName <> 'K';
```

You might think this would return the 10 rows that do not have K as a middle name, but that would be incorrect. Because the filter is looking for any value that does not equal K, it will discard any results that have null values. Nothing cannot equal (or even not equal) something, so the filter only considers rows that have a non-null value for MiddleName.

FirstName	MiddleName	LastName
Jorge	Armando	Guerra
Robert	Grant	Davidson
Gail	Anne	Shawn

Figure 7.9 The rows returned for any author that does not have a middle name of K, which excludes any author without a middle name

Of course, it is possible that your intention for this query is to only have rows with a value for MiddleName returned, but if you wanted all 10 rows returned, then you need to include the IS NULL operator in your filtering. You would use an additional OR operator, with the results shown in figure 7.10:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName <> 'K'
    OR MiddleName IS NULL;
```

FirstName	MiddleName	LastName
Doug	NULL	Li
Jen	NULL	Strong
Jorge	Armando	Guerra
Robert	Grant	Davidson
Gail	Anne	Shawn
Rebecca	NULL	Miller
Andy	NULL	Melkin
Buck	NULL	Fernandez
Chris	NULL	Walenski
Deepthi	NULL	Mahadevan

Figure 7.10 All rows that do not have a middle name of K are returned, including those with null values

7.2.3 How to search for values that are not null

Now that we've learned how to include rows with null values, let's look at how to return all rows that do not have a null value. We can start once again by declaring verbally what we want.

"I would like the first, middle, and last names of authors where the middle name isn't null."

Of course, the word *isn't* is a contraction for *is not*, which is exactly how our next operator will work, as shown by the following query and the results in figure 7.11:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName IS NOT NULL;
```

FirstName	MiddleName	LastName
Paul	K	Tripp
Jorge	Armando	Guerra
Robert	Grant	Davidson
Gail	Anne	Shawn

Figure 7.11 The results for all rows from the author table with no middle name

The IS NOT NULL operator allows us to return all rows with some value other than NULL for a given column. Interestingly enough, we can get the same results using a wildcard we learned earlier in this chapter:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName LIKE '%';
```

Why does this work the same as when we use the IS NOT NULL operator? Because the % wildcard will match any string of data, so long as there is data in the column. Because null values have no data, wildcards will never match them or return them in a result set, which is essentially what the IS NOT NULL operator also does.

TRY IT NOW

Execute the previous two queries using IS NOT NULL and the % wildcard and see how the results are the same as in figure 7.11

All right, we have spent the last three chapters examining a multitude of ways to filter results when querying a table. In the next chapter, we're going to level up your SQL knowledge even more by learning how to query multiple tables at the same time.

7.3 Lab

Let's take a moment to review the ways in which you have learned to filter rows. Write some SQL queries to find the following:

1. The full names of all authors who have a middle name of Anne or no middle name at all.
2. The full names of all authors who have no middle name and have a first name that starts with a D.
3. The title name and price of all titles that start with the word "The" and have a price less than \$10.00.
4. The title name and publication date of any title that ends with an S and was published after January 1st of 2020.
5. The title name of any title containing the word "of" or the word "in".

7.4 Lab answers

1. The answer is below:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName = 'Anne'
    OR MiddleName IS NULL;
```

2. The answer is below:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE FirstName LIKE 'D%'
    AND MiddleName IS NULL;
```

3. The answer is below:

```
SELECT
    TitleName,
    Price
FROM title
WHERE TitleName LIKE 'The%'
    AND Price < 10;
```

4. The answer is below:

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE TitleName LIKE '%s'
    AND PublicationDate > '2020-01-01';
```

5. This is a bit of a trick question designed to challenge you, because you have to consider that depending on how you write the query, you might get more or less data than you wanted. For example, you may write something as simple as this:

```
SELECT
    TitleName
FROM title
WHERE TitleName LIKE '%of%'
    OR TitleName LIKE '%in%';
```

If you execute that query, you will see that you not only get “The Call of the While,” “Anne of Fact Tables,” and “Catcher in the Try”—which are all meeting the requirement—but also “The Join Luck Club” and “The DateTime Machine,” which do not. The latter two are included in the results because they have a string value matching the value of the letters “in” within their title names.

So how can we exclude those undesired results? Well, one way would be to add leading and trailing spaces to the strings we are searching for, like this:

```
SELECT
    TitleName
FROM title
WHERE TitleName LIKE '% of %'
    OR TitleName LIKE '% in %';
```

This query will return the desired results, but this may not work in a different scenario. Consider that, since we are now looking at strings involving leading or trailing spaces, we will not have any results if the title names start or end with the words *in* or *of*. To include any hypothetical results that started or ended with the words we were searching for, we would need to include additional conditions, like this.

```
SELECT
    TitleName
FROM title
WHERE TitleName LIKE '% of %'
    OR TitleName LIKE '% in %'
    OR TitleName LIKE 'of %'
    OR TitleName LIKE 'in %'
    OR TitleName LIKE '% of'
    OR TitleName LIKE '% in';
```

Again, this was admittedly a difficult question, but it was designed to get you thinking about the possible values of data and how to creatively use what you’ve learned so far to write an accurate query.

8

Querying multiple tables

Back in Chapter 2, we discussed how relational database management systems (RDBMS) store data in objects known as tables, and since then we've been examining different ways to query these tables. I don't know if you've been wondering what makes a database management system "relational," but in this chapter we're going to answer that very question.

One of the primary features of an RDBMS is that it allows for a set of data to be stored so that it can relate to other sets of data, hence the use of the word *relational*. This way of storing data is incredibly powerful because we can not only put the data we have in logical groupings of tables but also easily retrieve related data from multiple tables with a single query.

Retrieving data in this way is done by *joining tables*, which means we combine the data in two tables using the values that form the relationship between the two tables. Although joining tables is very common and relatively easy, it does have some specific rules that must be followed to be done correctly. You'll soon learn about those rules and how to correctly write join tables in SQL, but before we do that, we need to consider a few vital concepts of relational databases and the way they are designed.

8.1 The rules of data relationships

Although we haven't focused on the words *relational* or *relationship*, we have already looked at one aspect of relationships in an RDBMS when we first started looking at rows in tables in Chapter 2. Think about a single row from any table: it is a collection of values that all relate to each other. For example, the first row in the title table contains values such as TitleID, TitleName, and others that all relate to the title "Pride and Predicates," so those values all relate to each other. It's the same for every row in the table, except that each row represents related values for a different title.

Although we haven't looked at any examples yet, values can also relate to other rows in other tables. We've taken all the information specifically related to a single title into the title table, but in the sqlnovel database we're going to have information that includes titles elsewhere. For example, one of the tables used to track information about orders of different titles is orderitem. Try the following query and take a look at the results in figure 8.1:

```
SELECT *
FROM orderitem;
```

OrderID	OrderItem	TitleID	Quantity	ItemPrice
1001	1	101	1	9.95
1002	1	101	1	9.95
1003	1	101	1	9.95
1004	1	101	1	9.95
1005	1	101	1	9.95
1006	1	101	1	7.95
1007	1	101	2	7.95
1008	1	101	1	7.95
1009	1	101	1	9.95
1010	1	102	1	9.95

Figure 8.1 The first ten rows in the orderitem table, which includes the column "TitleID"

The third column is clearly named TitleID, which is the same as the first column in the title table. This indicates that we have values in the orderitem table that relate to the values in the title table, which makes sense because the titles are what are being ordered by customers.

But why would these values be stored in different tables instead of just putting the title-related values we need in the orderitem table? Well, there are actually several good reasons for storing the data in separate tables, but rather than simply describing these reasons, it might be more helpful to provide you with an example that shows the reasons, by using some of the data we've already worked with.

8.1.1 Data without relationships

Suppose we designed a version of the sqlnovel database to track orders, and all the necessary data is stored in this single order table. We'll create this hypothetical table with columns for order date, title name, price, and customer's first and last names. It might look something like the results in figure 8.2:

OrderDate	TitleName	Price	FirstName	LastName
2015-06-01 00:00:00	Pride and Predicates	9.95	Chris	Dixon
2015-06-15 00:00:00	Pride and Predicates	9.95	David	Power
2016-09-02 00:00:00	Pride and Predicates	9.95	Chris	Dixon
2016-09-02 00:00:00	The Join Luck Club	9.95	Chris	Dixon
2017-11-14 00:00:00	The Join Luck Club	9.95	David	Power

Figure 8.2 shows our hypothetical table used to track orders that contains order date, title name, price, and customer name.

On the surface, this appears to be a logical way to track orders, and perhaps you have used a spreadsheet in a similar way. This may be fine for tracking a very small number of orders, but a closer look at the data reveals quite a bit of redundant values. In this table, we can see what appears to be orders for the same two titles placed by two different customers on different dates. The main problem isn't so much that we're using five rows of data to represent this, but rather that we have to repeat the data values so often.

Imagine this table having millions of rows, and you can see how over time this could become a problematic waste of query time and storage. This is especially true of the string values for title name and the customers' names because string values generally take much more storage space than numeric values.

That's not the only problem, though. What would happen if any values of data changed, such as the customer's last name? If a customer changed their name and placed a new order with their new name, how would we then connect the orders with the previous name to their new name? Well, we couldn't do it with this table, as they would appear as a different customer.

The same issue could exist if the data was entered incorrectly. How could we track sales if the last title name was inadvertently entered as "The Join Luck Clubs" with an extra s added to Club? We couldn't, as it would be a different value. Even though you or I can see it's just a mistaken data entry, in the data it would be a different title and it wouldn't show up in results if we wrote a SQL query that had WHERE TitleName = 'The Join Luck Club' as a filter.

As you can see, there are a lot of problems with storing all this data in a single table. Let's look at how we can use some basic relational database concepts to better organize this data.

8.1.2 Data with relationships

In a relational database, we want to eliminate as many of these redundant occurrences of the same values as possible. We can accomplish this by doing a few things.

1. We need to **organize the data into logical groups of values**. We then put these values into separate tables, and we want each row in each table to relate to something unique, such as the title of a book. Think again about how any given row in the title table contains data in this way.
2. We need to **determine what column or columns will contain a unique value in each of our new tables**. This column or set of columns will be known as the *primary key*, and it will be what allows us to relate data in other tables to this table. In the title table, this primary key is the TitleID column.

3. We then **replace the data in other tables with these primary key values** to represent the values we want to reference. Because these key values in our order table relate to values in another table, these will be known as *foreign keys*. In the case of the table in figure 8.2, we will replace the TitleName column with the corresponding TitleID values from the title table, since that is the key value.

Let's do this with the example of our order table. We can start by looking again at figure 8.2 and see how we can organize our data in this way.

First, we have the repeating values for TitleName, so it makes sense to create a separate table that stores these title values. The good news, as you have no doubt noticed, is that we already have a title name table in our sqlnovel database, which stores the data in this way. Let's look at the title table values for TitleID and TitleName for the two titles in our order table shown in figure 8.3:

```
SELECT
    TitleID,
    TitleName
FROM title
WHERE TitleName IN ('Pride and Predicates', 'The Join Luck Club')
ORDER BY TitleID;
```

TitleID	TitleName
101	Pride and Predicates
102	The Join Luck Club

Figure 8.3 The results for TitleID and TitleName for the titles “Pride and Predicates” and “The Join Luck Club.”

NOTE The values for TitleID in the title table must be unique so that we know exactly which row in the title table to reference. If the TitleID values are duplicated, the data becomes inconsistent and confusing, as we don't know which values are being referenced.

The TitleID column serves as the primary key, so we can replace the TitleName column in our order table with TitleID, which will correspond to values in the title table. In figure 8.4 we can see what our order table now looks like:

OrderDate	TitleID	Price	CustomerFirstName	CustomerLastName
2015-06-01 00:00:00	101	9.95	Chris	Dixon
2015-06-15 00:00:00	101	9.95	David	Power
2016-09-02 00:00:00	101	9.95	Chris	Dixon
2016-09-02 00:00:00	102	9.95	Chris	Dixon
2017-11-14 00:00:00	102	9.95	David	Power

Figure 8.4 shows what it would look like if we replaced TitleName with TitleID in our hypothetical order table.

We now have a relationship between these tables, which allows us to better avoid the problems we discussed earlier, at least as they relate to titles and their names. We're saving space by storing a smaller numeric value instead of a string for each time we want to refer to the title. We also now have less of an issue with data consistency, as there is a single source where the title name is stored. This means if any other tables want to reference a particular title name, they too can use the TitleID values.

This is essentially what makes an RDBMS so popular for storing so many kinds of data. Storing values in a relational way allows data to be stored efficiently, values to be changed easily, and, most importantly, the data to be consistent throughout the database.

In fact, if we look at our order table, we can consider other ways to store data more efficiently. Customers are unique individuals, and so any customer data should be in a separate table, with the data here being replaced with a key value. Again, we fortunately already have a customer table structured with these values and a primary key of CustomerID, as shown in figure 8.5 by the results of our query:

```
SELECT
    CustomerID,
    FirstName,
    LastName
FROM customer
WHERE (FirstName = 'Chris' AND LastName = 'Dixon')
    OR (FirstName = 'David' AND LastName = 'Power')
ORDER BY CustomerID;
```

CustomerID	FirstName	LastName
1	Chris	Dixon
2	David	Power

Figure 8.5 The results from the CustomerID, FirstName, and LastName for customers with the name “Chris Dixon” or “David Power”

Now we have three related tables, so let's replace the customer's name columns in our order table with a single column referencing their corresponding CustomerID values from our customer table. Our updated order table is shown in figure 8.6.

OrderDate	TitleID	Price	CustomerID
2015-06-01 00:00:00	101	9.95	1
2015-06-15 00:00:00	101	9.95	2
2016-09-02 00:00:00	101	9.95	1
2016-09-02 00:00:00	102	9.95	1
2017-11-14 00:00:00	102	9.95	2

Figure 8.6 shows our hypothetical order table with a CustomerID column to reference names in the customer table.

We now have a *one-to-many* relationship between the customer table and our order table. That means for each order, we have one customer, but any given customer can have more than one order. This is a very common type of relationship in relational databases.

Our data is organized even more efficiently, but there's one last change we can make. Consider the fourth and fifth rows in figure 8.6. It looks like Customer 1 purchased two different items on the same day, which for the purposes of our exercise are considered the same order. This makes sense, and we should expect there will be many times customers might order more than one item in a given order.

However, this poses a problem for creating a unique primary key on our order table, as we can't place a unique order key on the rows if there are duplicate rows for any given order. One common way to resolve this is to place the actual items ordered into a separate table, thus dividing the data related to an order into two tables. Because any given order can have one or more items included in the order, this also is considered a *one-to-many* relationship.

NOTE There are also *one-to-one* and *many-to-many* relationships between tables in relational databases. They are generally less common, so we're not going to look at any examples of those right now. Just know that there are other kinds of relationships between the tables of any given database.

If we're going to divide the data in our order table, we need to consider the following question for all the columns: Do the values relate to a specific item in the order, or generally to the order itself? Let's examine each column:

- OrderDate – These values are the same for the entire order, as all items are ordered at the same time in a given order.
- TitleID – The values are specific to an item, as an order can contain more than one title.
- Price – This value relates to individual titles, so it is also an item-level value.
- CustomerID – This value relates to the entire order, as the customer is the same for all items in an order.

Now that we've identified what values go into which tables, we can divide the values in our order table into two separate tables:

- orderheader, which contains the values unique to the entire order. We will create a primary key column for OrderID on the orderheader table.
- orderitem, which contains the values unique to the items in a given order. We will create a primary key column OrderItemID on the orderitem table, and a foreign key column of OrderID to create a relationship between orderitem and orderheader.

All right, now that we've organized our data efficiently, let's see how we can write SQL statements to join this data together.

8.2 The way to join data

Now that you have a basic understanding of how tables and keys are used, let's see how they are used in actual queries. We do this by focusing on the FROM clause in our queries, which is where we identify the data set to be used.

8.2.1 Joining two tables

If we wanted to find out which customer placed the first order, which is OrderID 1001, we might start with a query like this:

```
SELECT CustomerID
FROM orderheader
WHERE OrderID = 1001;
```

TRY IT NOW

You've waited long enough to write some SQL in this chapter, so go ahead and execute that query.

There's probably not a lot of value in showing a picture of a single column with a single value in the result set, so if you want to keep reading instead of executing the query, then know the value returned is 1.

Knowing the CustomerID for the first order is 1 might be useful for a lot of queries, but suppose we wanted to know the customer's name. For this, we need to utilize the relationship between the two tables by *joining* them in our query. We can do this by explicitly stating the second table name and the column names common to both tables, which we will use to join the data. We state all this in the FROM clause using some new keywords: JOIN and ON.

Here is a query that uses JOIN and ON to join the orderheader and customer tables to return not only the CustomerID but also the customer's first and last names, as shown in figure 8.7:

```

SELECT
    orderheader.CustomerID,
    customer.FirstName,
    customer.LastName
FROM orderheader
JOIN customer
    ON orderheader.CustomerID = customer.CustomerID
WHERE orderheader.OrderID = 1001;

```

CustomerID	FirstName	LastName
1	Chris	Dixon

Figure 8.7 The results of joining the orderheader and customer tables to show CustomerID and customer name values for the first order

Let's take a closer look at this query. The first thing you might notice is that the JOIN comes after FROM and the use of ON comes after that. The JOIN keyword is considered part of the FROM clause; it tells our RDBMS we want to use a second table.

Think of the usage of FROM and JOIN a bit like how we use WHERE and AND for filtering. If we have multiple filtering conditions for the WHERE clause, we start with the use of the keyword WHERE to state the first condition, but every subsequent condition will start with AND. Similarly, in the FROM clause, we start with FROM and then declare the first table we want data from, which in this query is the orderheader table. Then, because we also want data from a second table, we connect the data between the two tables by using the JOIN keyword, and then the name of the second table, which in this query is customer. Just as any subsequent filtering conditions would use the AND keyword, any subsequent table joins will use the JOIN keyword.

But merely stating that we want to JOIN the tables isn't enough, so we also need to say *how* we are relating these tables, by explicitly stating the columns we are using to establish the relationship. This is done with the use of the ON keyword in what is known as a predicate. A *predicate* is any part of our SQL statement that evaluates whether something is true, false, or unknown, which is what the ON section of the join is doing. It's finding all rows that match CustomerID in each table. If the match is true, then they are considered for inclusion in our result set.

Although it hasn't been mentioned previously, the filtering condition in the WHERE clause of our query is also considered a predicate, since we are asking the RDBMS to also evaluate WHERE orderheader.OrderID = 1001. Like that condition, our JOIN is considered an *exclusive condition*, so after evaluating the predicates in our join and our filtering conditions, only those that are considered true matches are returned. We only have one row in our results, because only those values meet all of our conditions.

Also notice that in the ON part of our query, we are using what is called *two-part names* for our columns. This name refers to the syntax of [table name].[column name] and it is crucial since the CustomerID column is included in both tables. We can't simply say ON CustomerID = CustomerID because the RDBMS won't know which CustomerID you mean. And if you think this is obvious in our example, be assured that many databases have tables with columns that relate to each other but have different column names. For this reason, we need to use two-part names.

TIP Although it doesn't matter which table and column is mentioned first in the ON portion of the JOIN, it's a good idea to start with the table mentioned first. This is for readability and organized data, of course, but as we will see in the next chapter, it will be crucial when working with other kinds of joins.

These two-part names for columns end up being used throughout the query, although that is mostly for readability. I say *mostly* because we could change almost all of them to use only the column name and not the table name, except for any time we use CustomerID. This is because the CustomerID column exists in both tables, so if our query said "CustomerID" anywhere without a reference to the table, we would get an error saying the "CustomerID" reference was ambiguous.

TRY IT NOW

Execute the query used to get the result in figure 8.7, but also try the query by changing orderheader.CustomerID in the SELECT clause to simply CustomerID and see the error that results in the Output panel.

8.2.2 Joining more tables

The great thing about joining tables is we aren't limited to just two tables. We can continue to use JOIN to connect more data, provided we know the correct columns used for the relationships between tables:

Recall how we have organized our order-specific data into two tables: orderheader and orderitem. If we wanted to find out even more information about that first order, such as the price of the item purchased in the order, we could easily add another join to our query. Since we established previously that the orderheader and orderitem tables will be joined on OrderID, which is the primary key for the orderheader table, we can modify our query with just a few more lines of SQL related to the orderitem table and see the results in figure 8.8:

```

SELECT
    orderheader.CustomerID,
    customer.FirstName,
    customer.LastName,
    orderitem.ItemPrice
FROM orderheader
JOIN customer
    ON orderheader.CustomerID = customer.CustomerID
JOIN orderitem
    ON orderheader.OrderID = orderitem.OrderID
WHERE orderheader.OrderID = 1001;

```

CustomerID	FirstName	LastName	ItemPrice
1	Chris	Dixon	9.95

Figure 8.8 Customer information from the customer table, and the price of the item ordered in the first order from the orderitem table

In this query, the JOIN for the orderitem table comes after the JOIN for customer, but in this particular query, the order of these joined tables doesn't really matter. We could just as easily have written the query with the JOIN for orderitem occurring before the JOIN for customer. The arrangement of the order of tables joined comes down to personal preference and readability, so long as both tables used in any JOIN have been declared in the FROM clause by the time you get to the ON portion of the join.

We can add one more table to get the title name of the item that was ordered, since we have the value for title name now in a fourth table: title. If you recall, the title name is now referenced in the orderitem table by the TitleID foreign key, which means we must include our JOIN to the title table after orderitem. Here's the query we will use, with the results shown in figure 8.9:

```

SELECT
    orderheader.CustomerID,
    customer.FirstName,
    customer.LastName,
    orderitem.ItemPrice,
    title.TitleName
FROM orderheader
JOIN customer
    ON orderheader.CustomerID = customer.CustomerID
JOIN orderitem
    ON orderheader.OrderID = orderitem.OrderID
JOIN title
    ON orderitem.TitleID = title.TitleID
WHERE orderheader.OrderID = 1001;

```

CustomerID	FirstName	LastName	ItemPrice	TitleName
1	Chris	Dixon	9.95	Pride and Predicates

Figure 8.9 Customer information from the customer table, the price of the item ordered in the first order from the orderitem table, and the title name from the title table

We can keep joining more tables to get related data, and in later chapters, as we discuss more of the tables in our database, we will do just that. But as you can see, our queries with all these two-part names make for a lot of words in our SQL query. Even for seasoned query writers, this is a wordy way to write a query with a few joins. There is a much more readable way to write these two-part names, and it involves a familiar concept.

8.3 Table aliases

Recall that in Chapter 3 we talked about renaming columns in our result set by using *aliases*. We effectively declared that a column would be referenced, at least in the output, but with a different name. Fortunately, the SQL language allows us to use aliases for table names as well.

Our goal in using table aliases is different from our usual goal with column aliases. With columns, we often want to change the column name in the results to be more descriptive, but with table aliases we want to be less descriptive. We generally will use an alias of one or two characters, because this will reduce the overall number of characters in our query, which, if done correctly, can make our query easier to read.

One common way to alias the table names is to use one- or two-letter abbreviations for the tables that are being aliased. For example, we can use an alias of “c” for the customer table or “t” for the title table. We could use an “o” as an alias for the orderheader table, but since another table in our query starts with an o (orderitem), we can use two-letter aliases for those tables instead. One logical way to use an alias for these tables would be to use the first letter of each word in the table names, like “oh” for orderheader and “oi” for orderitem.

Here is an example of these types of aliases, using the previous query:

```
SELECT
    oh.CustomerID,
    c.FirstName,
    c.LastName,
    oi.ItemPrice,
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderID = 1001;
```

There are definitely fewer characters filling the screen for that query, which means we have less information to review if we want to understand this query. Of course, we could also alias these table names the same way we aliased columns if we wanted to. For example, we could alias `orderheader` by saying `FROM orderheader AS oh`, but this isn't often done since our goal with aliasing table names is to reduce the overall number of characters used in our query. For this reason, it's highly encouraged to use table aliases as shown here when joining tables in any query, as it makes it much easier to read.

There are a few rules we have to follow, however, for table names. First, aliases must start with an alphabetical character, and not a number or a special character. Also, other than the first character, an alias can contain numbers but not special characters. Besides these rules, the only other consideration for aliases is to make them sensible, and not just alias the first table as `a`, the second table as `b`, and so on. If the aliases are at least remotely representative of the actual table names, your SQL query will be much easier to read and understand.

TRY IT NOW

Choose any of the queries we've used in section 8.2 and rewrite them using your own aliases.

8.4 The other way to join data

Earlier in the chapter we discussed predicates, and how they evaluate conditions in both the joins we use in our `FROM` clause and the conditions we state in the `WHERE` clause. Although it is rarely used, there is actually a way to combine all the predicates in the `WHERE` clause.

This is only being mentioned because at some point you will likely encounter a SQL query written by someone else that uses this methodology for joins. As you'll soon see, it's generally discouraged for several reasons.

Here's what our previous query would look like using this other way of joining:

```
SELECT
    oh.CustomerID,
    c.FirstName,
    c.LastName,
    oi.ItemPrice,
    t.TitleName
FROM
    orderheader oh,
    customer c,
    orderitem oi,
    title t
WHERE oh.OrderID = 1001
    AND oh.CustomerID = c.CustomerID
    AND oh.OrderID = oi.OrderID
    AND oi.TitleID = t.TitleID;
```

The first thing that might stand out to you is that with this method of joining data, we have an easily readable, comma-separated list of tables in the FROM clause. This is a bit closer to the verbal English we've been considering throughout the book, because we might verbally declare the intentions for this query something like this:

"I would like the customer ID, first name, last name, item price, and title name from the orderheader table, customer table, orderitem table, and title table where the Order ID is 1001."

But even this method is difficult to convert from a verbal declaration to SQL since we have to tell the RDBMS exactly how all those tables need to be joined. Although this is a perfectly valid way to join data in SQL, it does come with several disadvantages.

First, when it comes to finding exactly how all these tables are joined together, we have to read every row in the WHERE clause to determine how any single join is evaluated. This may be using fewer characters since we aren't saying JOIN for each join, but for this query and other, more complex queries, we have to scan the entire WHERE clause to find how any two tables are joined. Combining all these evaluations in the WHERE clause makes troubleshooting much more difficult—like trying to find a particular noodle in a bowl of spaghetti.

Second, this method only allows for a particular type of join. In the next chapter, we will be discussing various ways to use JOIN to connect data in more inclusive ways. We won't be able to connect our data any other way with this method. For these reasons, you should avoid writing SQL that contains joins in the WHERE clause.

Joins will be a critical component of nearly every query you will write from now on, so if you feel unsure about how they work, then please take time to review this chapter and practice the query examples starting in section 8.2, as well as the following lab exercises. If you're feeling confident about joining tables, then great – we'll see you in the next chapter!

8.5 Lab

- What is the difference in the output of the following two queries, where you use a different table for filtering in the WHERE clause?

```
SELECT
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderID = 1001;
```

```
SELECT
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oi.OrderID = 1001;
```

- How many orders did the customer named Chris Dixon place? Write a query to determine the answer.
- What are the names of the customers who ordered a title in 2015? Write a query to determine the answer.
- How could you rewrite the following query, which finds the names of all customers who ordered "The Sum Also Rises," using JOINs and aliases?

```

SELECT
    customer.FirstName,
    customer.LastName
FROM title, orderheader, customer, orderitem
WHERE title.TitleName = 'The Sum Also Rises'
    AND orderheader.OrderID = orderitem.OrderID
    AND orderitem.TitleID = title.TitleID
    AND orderheader.CustomerID = customer.CustomerID;

```

5. We saw in section 8.4 how we can move all the predicates to the WHERE clause. What happens if we move all the predicates to the FROM clause, as in the following query?

```

SELECT
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
    AND oh.OrderID = 1001
JOIN title t
    ON oi.TitleID = t.TitleID;

```

8.6 Lab answers

- The results will be the same. Since our query is matching all OrderID values from the orderheader to the orderitem table, the OrderID column from either table can be used in the filtering condition to return the same results.
- Chris Dixon placed 3 orders. You could use a query like this to get the results:

```

SELECT
    oh.OrderID
FROM orderheader oh
JOIN customer c ON oh.CustomerID = c.CustomerID
WHERE c.FirstName = 'Chris'
    AND c.LastName = 'Dixon';

```

- Eight customers placed an order in 2015:
Chris Dixon

David Power
 Arnold Hinchcliffe
 Keanu O'Ward
 Lisa Rosenqvist
 Maggie Ilott
 Cora Daly
 Dan Wilson
 You could find them with a query like this:

```
SELECT
    c.FirstName,
    c.LastName,
    oh.OrderDate
FROM orderheader oh
JOIN customer c ON oh.CustomerID = c.CustomerID
WHERE oh.OrderDate >= '2015-01-01 00:00:00'
AND oh.OrderDate < '2016-01-01 00:00:00';
```

4. You could write a query like this using JOINs and aliases:

```
SELECT
    c.FirstName,
    c.LastName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE t.TitleName = 'The Sum Also Rises';
```

5. The query executes successfully with the same result set, but as noted in this chapter, we generally try to avoid combining filtering predicates with join predicates for the sake of readability.

9

Using different kinds of joins

Joining tables is an essential skill to have when writing SQL queries, but so far, we have tried only one kind of join. To be fair, that type of join is the most common, but as you'll see in this chapter, there will be plenty of scenarios where that kind of join won't help you produce the results you need.

For example, you may be asked to produce a list of all orders for a given year and show if they used a particular discount code, or to find the names of all customers who didn't place an order in a year, or to find a list of all customers in a particular city or state and show which placed orders and which did not. None of these queries can be accomplished using the type of join from the last chapter.

In this chapter, we're going to learn how to use different joins in SQL to fulfill all the preceding requests. Let's dive into these types of joins and see how to use them.

9.1 Inner joins

First, let's talk a little bit more about the JOIN keyword we used in the previous chapter. This is actually a shorthand version of the keywords INNER JOIN, which is a particular type of join. Because it only joins values in both tables that meet the conditions of the join, the results set *excludes* any rows that do not meet the conditions.

For much of this chapter, we're going to use two tables: the orderheader table and a different table named promotion. This promotion table contains promotion codes that can be used for discounted prices on titles that are ordered. The key in the promotion table is PromotionID, and it is referenced by a similarly named PromotionID column in the orderheader table.

The reason we will be using these tables is that there are rows in both tables which do not relate to the other. This is to say, there are rows in the promotion table that represent promotions that were never used in any order, and there are rows in the orderheader table that were placed without using a promotion code. Also, the relationship between these tables is one-to-many, as any promotion code can be used for more than one order, but every order can use only one promotion code.

Additionally, we should note there are 12 rows in the promotion table and 50 rows in the orderheader table. We will refer to the number of rows in these tables throughout the chapter.

TRY IT NOW

Use `SELECT * FROM promotion` and `SELECT * FROM orderheader` to see how many rows are returned. Check the Message in the Output panel to confirm the number of rows in each table. If you prefer, you could also count the rows returned yourself, but that's more time-consuming, as well as open to the possibility of human error.

Let's start with the following query, which will find the order ID and promotion code of any order that used a promotion code, with a portion of the results shown in figure 9.1:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

OrderID	PromotionCode
1006	ZOFF2015
1007	ZOFF2015
1008	ZOFF2015
1013	ZOFF2016
1014	ZOFF2016
1015	ZOFF2016
1016	ZOFF2016
1022	ZOFF2017

Figure 9.1 A portion (8 of the 20 rows) of the results showing the order ID and promotion codes of all orders that used a promotion code

Looking at the Output window, we can see a message that reads “20 row(s) returned.” You might also notice the values for PromotionCode are not 20 unique promotion codes, but the values for OrderID are 20 unique values. Again, this is because of the one-to-many relationship between the tables, where each promotion code may be used for multiple orders.

If we wanted to be more verbose, we could write the same query by describing our join explicitly as an INNER JOIN, returning the same 20 rows:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

TIP If you are only using inner joins in a particular query, then it's acceptable to just write JOIN instead of INNER JOIN. However, if a query will include other joins, such as those you are about to learn, you should specify INNER JOIN for clarity and readability.

A common way to consider how the values in these tables logically relate to each other is through a Venn diagram. Consider two intersecting circles, where each circle includes the data of a single table. The intersecting parts of the circles represent the common values between the two tables, and the non-intersecting parts represent the values unique to each table.

Here is a Venn diagram of the inner join between the promotion and orderheader tables. We will look at several Venn diagrams throughout this chapter to help us visualize the data included in the different types of joins. The colored part of the data in figure 9.2 represents the data that is returned in our result set, and the empty parts represent the data omitted from our results.

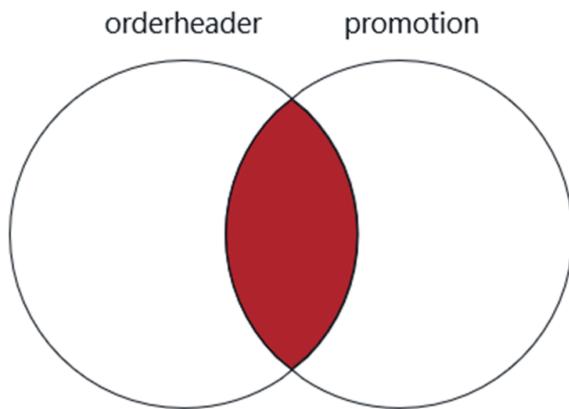


Figure 9.2 A Venn diagram of the inner join used in the previous query

The data returned by our inner join only includes the common values that meet the condition of our join, where the PromotionID values in each table match. As mentioned, this is not our only option when joining tables. We also have options to join the tables in such a way as to include all the values of one of the tables that is joined, even if the rows have no related values in the other table. This is done by using OUTER JOIN keywords.

9.2 Outer joins

The syntax used for outer joins is similar to that used by inner joins, in that you use the JOIN keyword to specify the tables you are joining and the ON keyword to identify the condition of the columns that are used in the relationship between the tables. If necessary, additional conditions for the join will use the AND keyword.

One big difference between inner and outer joins is there are different types of outer joins, so you need to state in your SQL which type of outer join you are using. Let's start with the LEFT OUTER JOIN.

9.2.1 Left outer joins

The use of the word *left* in LEFT OUTER JOIN indicates we want all rows returned from the *left* table in our join, regardless of whether they match. If this seems confusing, think of it as also meaning we want all the rows from the *first* table noted in our join.

Suppose we would like to see a list of all order IDs, and if they used a promotion code, then we want to see which promotion code was used. We would use the same query as before, but we would change our INNER JOIN to a LEFT OUTER JOIN:

```

SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;

```

In this query, the left table is `orderheader`, since it is the first table mentioned. In our formatted query, `orderheader` appears above `promotion` and not to the left, but if this was contained on one unformatted line with no carriage returns, then `orderheader` would be to the left of `promotion` in the query.

We would represent this kind of left outer join with a diagram like the one in figure 9.3.

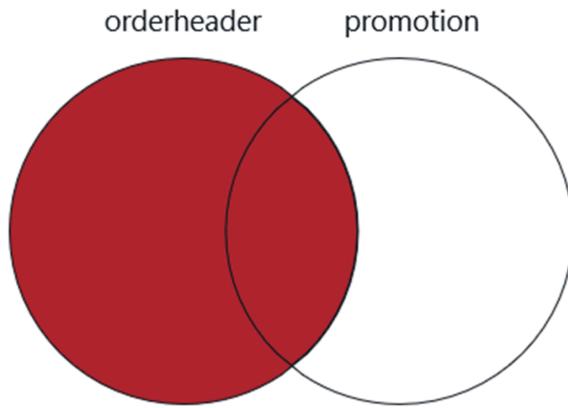


Figure 9.3 A Venn diagram of the left outer join

The message in the Output panel lets us know this query returns 50 rows, which is to be expected since we earlier learned that's how many rows are in the `orderheader` table. These 50 rows are a lot to take in, so let's add a filter to limit the results to the first eight orders placed, as shown in figure 9.4:

```

SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.OrderID <= 1008;

```

OrderID	PromotionCode
1001	HULL
1002	HULL
1003	HULL
1004	HULL
1005	HULL
1006	ZOFF2015
1007	ZOFF2015
1008	ZOFF2015

Figure 9.4 shows the order ID and any promotion codes used for the first eight orders. Promotion codes were used only for order IDs 1006, 1007, and 1008.

The result set includes a row for every row in the orderheader table that meets our condition of being less than or equal to OrderID 1008, regardless of whether they have a PromotionID value to join to the promotions table. For those orders that do not have a promotion code, the results show a null value in the PromotionCode column.

Now, although we've done a lot of work with filtering so far, we need to be careful when adding filtering conditions to outer joins. If we add a filtering condition for a specific value in the right table in the WHERE clause of our query with a LEFT OUTER JOIN, then we effectively turn our LEFT OUTER JOIN into an INNER JOIN. This is because filtering on specific values in the right table would eliminate any rows from our result set which might have null values in the right table. We can demonstrate this with a similar query that filters on a particular value for the promotion code, with the results shown in figure 9.5:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE p.PromotionCode = 'ZOFF2015';
```

OrderID	PromotionCode
1006	ZOFF2015
1007	ZOFF2015
1008	ZOFF2015

Figure 9.5 shows the results of the left outer join with a filter on the right table (promotion), which reduces our result set to one that would be the same as if this were an inner join.

The results now include only three rows for orders that used the promotion code 2OFF2015, and not a row for every OrderID in orderheader as we would expect from a left outer join. Since the condition in the WHERE clause was required of the values in the right table, which in this query is promotion, that condition is applied to the results from both tables. Because of this filtering condition on the right table in the WHERE clause, we could change the LEFT OUTER JOIN in our query to INNER JOIN instead and get the same results.

TRY IT NOW

Execute the previous query for a different promotion code, such as 2OFF2016. Execute it once with a LEFT OUTER JOIN, and again with an INNER JOIN to see how the results are the same.

If we truly wanted a list of all orders, and to see if they used a specific promotion code such as 2OFF2015 instead of any promotion code, we could still do this. We just need to move the filtering from the WHERE clause to the join condition, like this:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
    AND p.PromotionCode = '2OFF2015';
```

This query will return 50 rows, one for each order ID since we are not filtering on the left table. The value for the PromotionCode column in the result set will be either 2OFF2015 or null.

Please make a note of how this filtering in the join condition works, because there will be many occasions when you need to use an outer join while also filtering on specific values in two or more tables. If you filter in the WHERE clause for a table joined with an outer join, then you may inadvertently create an inner join.

9.2.2 Right outer joins

Just as the LEFT OUTER JOIN will return all rows in the left/first table, regardless of whether they match the right/second table, the RIGHT OUTER JOIN will do the opposite. It will return all rows in the *right* table whether or not they match rows in the left table with the join condition.

Let's use a right join to show all promotional codes regardless of their usage, with a portion of the results shown in figure 9.6:

```

SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
RIGHT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;

```

PromotionCode	OrderID
ZOFF2015	1008
ZOFF2015	1007
ZOFF2015	1006
ZOFF2016	1016
ZOFF2016	1015
ZOFF2016	1014
ZOFF2016	1013
ZOFF2017	1024

Figure 9.6 A portion (8 of the 23 rows) of the results showing all promotion codes, and order IDs if they were used in a promotion

We can visually represent the results of our right join with the diagram shown in figure 9.7.

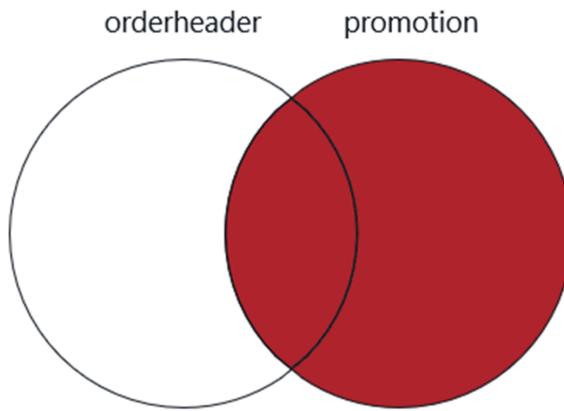


Figure 9.7 A Venn diagram of the right outer join

The previous query returns 23 rows, which is more than the 12 rows in the promotion table. Look closely and you will see that many of the rows include duplicate values for PromotionCode, since a code can be used for more than one order. Because of the duplicate usage of certain promotion codes, we have matched many orders to some of the promotion codes.

Scroll through the results and you will also see that a few of the PromotionCode values have null for OrderID. This is because those promotion codes were not used in a corresponding order, so they didn't match any order. We have them in our result set anyway, since all rows in the promotion table, which is the right table in our query, will have at least one row in our result set from the right outer join.

9.2.3 Using outer joins to find rows without matching values

Just as we can use a left or right join to return all rows in a table regardless of whether they match, we can also use either kind of outer join to find all the rows that don't match. We do this by explicitly saying we want to find rows in the matching table where the filter condition is null.

As an example, we can write a query to show us only the promotion codes that have not been used for any orders, as shown in figure 9.8, by adding the filter WHERE oh.PromotionID is NULL to our previous query, like this:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
RIGHT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.PromotionID IS NULL;
```

PromotionCode	OrderID
1OFF2020	NULL
1OFF2021	NULL
2OFF2021	NULL

Figure 9.8 The results for all promotion codes that do not have a corresponding order ID, meaning the promotion code was never used

Although previously it was noted that placing a filtering condition on the joined query will effectively turn an outer join into an inner join, checking for null values is the exception. Remember, checking for a null value isn't checking for an equality between two values, but rather we are querying for the presence of null values. This is a common kind of query, as you will often have to find some value that exists in one table but does not exist in another table. We can represent this concept with a diagram like the one in figure 9.9.

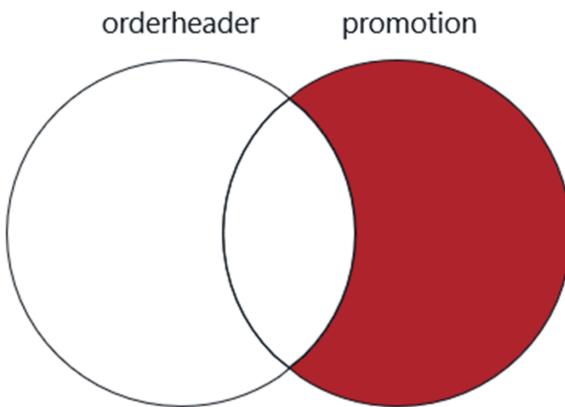


Figure 9.9 A Venn diagram of the right outer join that excludes rows from the left table

9.2.4 Interchanging left and right joins

If you try to execute the previous query in SQLite, it won't work because that RDBMS doesn't support the RIGHT OUTER JOIN command. However, this won't be a problem for most queries, as you could simply rewrite the join as a LEFT OUTER JOIN instead, like this:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
LEFT OUTER JOIN orderheader oh
    ON p.PromotionID = oh.PromotionID
WHERE oh.PromotionID IS NULL;
```

This query will produce the same results as those shown in figure 9.8 because all we have done is swap the order of two tables in the FROM clause and change the join from right outer to left outer. Here's a diagram in figure 9.10 to show what we just did:

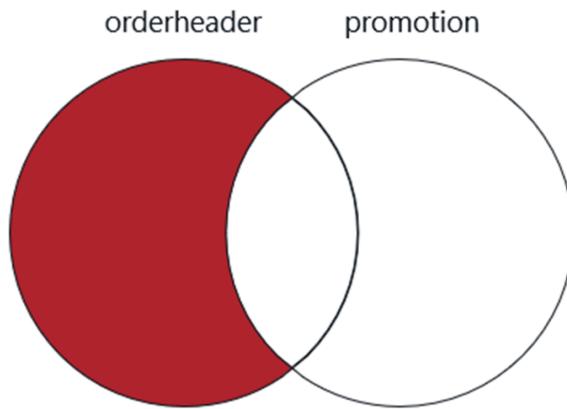


Figure 9.10 A Venn diagram of the left outer join that excludes rows from the right table

TIP As much as possible, try to use only left or only right outer joins in any query, and not both. There are very few cases where you need to include both types of outer joins, so using only one type of outer join will make your query easier for others to understand. In fact, some RDBMSs do not support right outer joins at all. For this reason, left joins will be preferred throughout the remainder of this book.

One last note about left and right joins is that they don't need to be quite so verbose. Just as INNER JOIN can be shortened to JOIN, LEFT OUTER JOIN and RIGHT OUTER JOIN can be shortened to LEFT JOIN and RIGHT JOIN, respectively. For example, the previous query can be written without the OUTER keyword, which may improve readability:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
LEFT JOIN orderheader oh
    ON p.PromotionID = oh.PromotionID
WHERE oh.PromotionID IS NULL;
```

NOTE Depending on your RDBMS, you may also have the ability to use a FULL OUTER JOIN or FULL JOIN. This rarely used type of join returns the combined results of both a LEFT JOIN and RIGHT JOIN of two tables. We won't be writing any queries with FULL OUTER JOIN since MySQL, Maria DB, and SQLite do not support this type of join, but you will learn another way to produce this kind of result set in the next chapter.

9.2.5 The USING keyword

There are two other ways to write inner, left, right, and outer joins, but they are less common. The first way is with the USING keyword. The USING keyword will replace the ON keyword in the join, and it does not require specifying the table names or aliases in the join. In fact, you can't specify the table names or aliases, since the USING keyword requires the names of the columns used in the relationship to be the same in both tables. We could rewrite the previous query with USING to find the results shown in figure 9.8 like this:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
LEFT JOIN orderheader oh
    USING (PromotionID)
WHERE oh.PromotionID IS NULL;
```

The requirement of the column names being identical for a join is usually the biggest deterrent to using, well, USING, as you will encounter many databases with related tables that do not share the same column names. The USING command isn't frequently used, and many other SQL programmers aren't aware of it or its correct usage. We're only looking at it here in case you notice it in someone else's SQL queries.

9.2.6 Natural joins

The second rarely used way to write inner, left, right, and outer joins is with something called a natural join. With a *natural join*, you don't even mention the column names involved in the relationship as they, seemingly by magic, join columns of the same name from two tables. This is done by adding the NATURAL keyword while altogether omitting the usage of ON or USING. Let's rewrite the previous query one more time, this time with a natural join:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
NATURAL LEFT JOIN orderheader oh
WHERE oh.PromotionID IS NULL;
```

Although the amount of SQL written is reduced further, it is highly recommended to avoid using natural joins. For starters, they don't identify the columns used in the relationship between the tables—so whoever reads your SQL will have no idea how the promotion and orderheader tables are related to each other.

But the bigger problem involves similarly named columns. While our promotion and orderheader tables share only a single, easily identifiable column with the same name, in real-world scenarios, you will see many tables that contain columns like “CreateDate” or “ModifiedDate” to track changes in the values of any given rows. Although these are often similarly named columns in a given database, they clearly are not created for the purpose of relating data. Using a natural join with tables containing these commonly named columns would then automatically join the data on those columns, which would definitely not provide the results expected.

WARNING Natural joins are not supported in SQL Server.

9.3 Cross joins

The last kind of join to look at is an unusual one: the cross join. The thing that makes this type of join unusual is that, unlike all the other joins we've discussed, it isn't used to find rows with specific values. Rather, a *cross join* finds all possible combinations of rows by matching every row from one table to every row in another.

The cross join is also known as a *Cartesian join*, since the results of the cross join reflect the mathematical operation known as a Cartesian product, which describes this result set of all possible paired values from two sets of data. If we wanted to use a cross join to show all possible combinations of promotion codes from the promotion table, and order IDs from the orderheader table, then we could write a query like this:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
CROSS JOIN orderheader oh;
```

The results of this query reflect all possible combinations of matching the values from the two tables, so it should be no surprise that the result set for this query is 600 rows. This is because we have 12 rows in the promotion table and 50 rows in the orderheader table. A little multiplication of 12×50 confirms that 600 rows are to be expected in the results.

While a cross join isn't helpful for finding particular rows, it is essential for times when you may need to generate a full list of all possible outcomes. For example, you might need to produce a grid for all sizes and colors of a particular product. It can also be helpful for quickly generating a lot of test data, such as a list of customers or orders that are much larger than your current data contains.

WARNING Although we previously noted that you don't need to specify an inner join with the word INNER in MySQL, if you join tables using only the word JOIN and omit a join condition, then the results will reflect a cross join and not an inner join as intended. Other RDBMSs may require ON when using INNER joins.

The cross join is the last of many types of joins we've examined in this chapter. It may seem discouraging that several of the shorter ways to write SQL joins are not recommended, but in the next chapter we'll look at other ways to join data using more efficient methods that are more highly accepted and sometimes more efficient for the RDBMS. For now, let's practice what you've already learned in a few lab exercises.

9.4 Lab

At the beginning of this chapter, we discussed some possible scenarios that you might encounter. Let's start by using what you've learned to write queries to produce those results. Try to use a left join in each of the first three exercises.

1. Write a query that shows the order ID and order date of all orders from 2019, and also the promotion code if one was used.
2. Write a query to show the first and last names of all customers who did not place an order in 2020.
3. Write a query to show the first and last names of customers as well as the order ID and order date for any orders placed in 2021 by customers in California (where the value for State in the customer table is CA).
4. This exercise wasn't mentioned at the beginning of the chapter, but write a query using a cross join to generate a list of all possible customer first names from the customer table and last names from the author table.

9.5 Lab answers

1. The answer is below:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.OrderDate >= '2019-01-01'
    AND oh.OrderDate < '2020-01-01';
```

2. The answer is below:

```
SELECT
    c.FirstName,
    c.LastName
FROM customer c
LEFT JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
    AND oh.OrderDate >= '2021-01-01'
    AND oh.OrderDate < '2022-01-01'
WHERE oh.CustomerID IS NULL;
```

3. The answer is below:

```
SELECT
    c.FirstName,
    c.LastName,
    oh.OrderID,
    oh.OrderDate
FROM customer c
LEFT JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
    AND oh.OrderDate >= '2021-01-01'
    AND oh.OrderDate < '2022-01-01'
WHERE c.State = 'CA';
```

4. The answer is below:

```
SELECT
    c.FirstName,
    a.LastName
FROM customer c
CROSS JOIN author a;
```

10

Combining queries with set operators

In the last few chapters, we've examined different ways to join tables based on the way they relate to each other. Every query we've written until now has had a single SELECT statement, but in this chapter we're going to see how you can write a query with multiple SELECT statements and combine the results into a single set of data.

This can be useful when we need to evaluate results that require different conditions, such as querying values in different tables with no key to join the two tables. And although we've already seen how null values are excluded from results using joins, we are going to see how we can use SQL to include null values if they exist in two sets of data and we want them included in our results.

10.1 Using set operators

We've written a lot of queries so far that start with SELECT, and every one of them results in a single result set. Remember, that's what SELECT queries do: they produce a set of results. More specifically, they produce a set of rows that meet the various conditions of our queries.

There will be times though when you want to combine or evaluate two or more result sets, and to do this we need to use special keywords known as *set operators*. Though there aren't many set operators in SQL, they all use the same syntax to evaluate two result sets. Here's what that looks like:

```

SELECT <some column>, <another column>
FROM <some table>
WHERE <some condition>
<set operator>
SELECT <some column>, <another column>
FROM <some table>
WHERE <another condition>;

```

Even though we are using two different SELECT statements in our query, the set operator will allow us to evaluate the results into a single result set. The most common evaluation is to combine them, but as you'll see later in this chapter, we can do more than that.

There are a few rules to using set operators, however, that must be adhered to. They are:

THE NUMBER OF COLUMNS NEEDS TO MATCH

This is the most obvious rule, as we already know from our introduction to tables that every row in a table must have the same number of columns. Our result set using a set operator is no different, and attempting to evaluate queries with a different number of columns will result in an error.

THE DATA TYPE OF EACH COLUMN NEEDS TO MATCH

We haven't talked too much about data types yet, but we have seen how there are different data types for numbers, characters, and dates. If you attempt to combine different data types in a result set, then you will receive an error message.

THE NAMES OF THE COLUMNS IN THE FIRST QUERY WILL BE USED IN THE RESULT SET

This means you can evaluate columns with different names, but their ordinal positions must be the same in each SELECT. If you are going to use column aliases, only those used in the first SELECT will apply to the results. You can certainly add column aliases to any SELECT statement other than the first one in your queries without causing an error. Just remember they will be ignored by the RDBMS and not affect the result set. Understanding this rule is also important because of the last rule, described next.

IF YOU ARE USING AN ORDER BY CLAUSE, THEN IT MUST APPEAR IN THE FINAL SELECT STATEMENT

An ORDER BY is the last evaluation in any query, and as such it is only allowed after the last of our SELECT statements. If you try to sort the results in any other SELECT statement, you will receive an error message.

Now that you know the rules, let's look at some ways to use set operators.

10.2 Union

The most common set operator is UNION, which allows you to combine the results of two or more SELECT statements into a single result set, removing any duplicates.

TIP One of the most important things to remember about UNION is that it removes duplicate rows from your result set. Do not forget this.

As an example, we can combine the names of all the people in our sqlnovel database into a single set of names. This means we will select the first and last names from both the customer and the author tables into a single list of names. Let's select the names from both tables and order by last name and first name, with the results shown in Figure 10.1.

```
SELECT FirstName, LastName
FROM customer
UNION
SELECT FirstName, LastName
FROM author
ORDER BY LastName, FirstName;
```

FirstName	LastName
Sandra	Calderon
Cora	Daly
Kevin	Daly
Robert	Davidson
Tara	Di Silvestro
Chris	Dixon
Jordan	Ericsson
Buck	Fernandez

Figure 10.1 A portion (8 of the 31 rows) of the results of first and last names from the customer and author tables

The results of our two SELECT statements have been combined into a single result set and then ordered as we've directed in figure 10.1. But why the word UNION? Well, if we wanted to verbally declare what we are requesting with our SQL, we could say something like this:

"I would like the first and last names from the customer table, and I would like to combine the results with the first and last names from the author table."

Although the word *combine* accurately describes what we are doing, as you'll see throughout this book there are all sorts of ways to combine things in SQL. We can combine rows, columns, values, and entire result sets in different ways, so the word *combine* isn't specific enough to what we are requesting.

Instead, we use the word *union* to describe combining two or more sets of data into a single result. In English, the verb *union* often describes marrying people from two different families into one new family, so it might be helpful to think of the UNION operator as marrying two different sets of data into a single result set. Our verbal declaration becomes a bit more descriptive with the word *union*:

"I would like the first and last names from the customer table, and I would like to union the results with the first and last names from the author table."

Although we can't inherently know whether any given row was selected from the customer table or author table, we can verify the table of origin by adding a third column with literal values to indicate the table where the rows are from. We will add these literal values to both SELECT statements, but we only need to add the column name to the first SELECT statement as shown in figure 10.2. As noted earlier, the column names in the result set are chosen from the first SELECT statement:

```
SELECT FirstName, LastName, 'customer' TableName
FROM customer
UNION
SELECT FirstName, LastName, 'author'
FROM author
ORDER BY LastName, FirstName;
```

FirstName	LastName	TableName
Sandra	Calderon	customer
Cora	Daly	customer
Kevin	Daly	customer
Robert	Davidson	author
Tara	Di Silvestro	customer
Chris	Dixon	customer
Jordan	Ericsson	customer
Buck	Fernandez	author

Figure 10.2 shows a portion (8 of the 31 rows) of the results of first and last names from the customer and author tables, as well as a third column indicating the table where the rows are from.

The most common way of using a union is for combining various filtering conditions, especially ones that might be contradictory in a single SELECT statement, such as values from different tables like we are querying. Let's put some filtering conditions for LastName from the customer table and FirstName from the author table:

```

SELECT FirstName, LastName, 'customer' TableName
FROM customer
WHERE LastName LIKE 'D%'
UNION
SELECT FirstName, LastName, 'author'
FROM author
WHERE FirstName LIKE 'C%'
ORDER BY LastName, FirstName;

```

FirstName	LastName	TableName
Cora	Daly	customer
Kevin	Daly	customer
Tara	Di Silvestro	customer
Chris	Dixon	customer
Chris	Walenski	author

Figure 10.3 The results for the full names of customers whose last name starts with D and authors whose last name starts with C, ordered by last name and first name

In figure 10.3 we now have only five rows that meet the filtering criteria, and we can see we have at least one row from each table. Notice how there are duplicate values for first name ("Chris") and last name ("Daly"), and recall that we noted how duplicate rows where all values match are removed when using UNION. We can verify this with a few changes to our query.

Two rows have the first name of "Chris" in figure 10.3, of which there is one in each of the tables we are querying. Let's omit the LastName and TableName columns from our results, since those extra columns create unique rows for the rows that have "Chris" as a value for FirstName. With these rows omitted, we should expect there to be only one row represented for "Chris," since duplicates are removed.

In addition to omitting those columns, we will change the ORDER BY from LastName to FirstName. When using a set operator like UNION, we are only able to sort the results by columns included in the SELECT clause. We have removed LastName from the SELECT statement, so if we left the SQL for ordering by LastName in our next query, then we would get an error on execution indicating LastName was an "Unknown column."

Here's our new query:

```

SELECT FirstName
FROM customer
WHERE LastName LIKE 'D%'
UNION
SELECT FirstName
FROM author
WHERE FirstName LIKE 'C%'
ORDER BY FirstName;

```

FirstName
Chris
Cora
Kevin
Tara

Figure 10.4 shows the results of first names from the customer table whose last names start with D, combined with a UNION with the results of first names from the author table whose first names start with C. The two rows for “Chris” are represented with a single row since UNION has removed the duplicate from the result set.

But what if we didn’t want the duplicate rows removed, and instead wanted any duplicate rows to be *included* in a result set? For that scenario, we can look at the next set operator, UNION ALL.

10.3 Union All

The UNION ALL set operator is very much like the UNION operator, but with the main difference of not removing duplicate rows. Instead, UNION ALL instructs the RDBMS to read all the data as requested by each SELECT statement and return the results as they were read.

We can modify the set operator in the previous query from UNION to UNION ALL to see this difference:

```
SELECT FirstName
FROM customer
WHERE LastName LIKE 'D%'
UNION ALL
SELECT FirstName
FROM author
WHERE FirstName LIKE 'C%'
ORDER BY FirstName;
```

FirstName
Chris
Chris
Cora
Kevin
Tara

Figure 10.5 shows the results of first names from the customer table whose last names start with D, combined with a UNION ALL with the results of first names from the author table whose first names start with C.

The two rows for “Chris” are both included in the results in figure 10.5 because UNION ALL does not remove duplicate rows from the result set. Since it doesn’t remove duplicates, the filtering conditions used by UNION ALL can function similarly to the filtering we do in a WHERE clause. For example, there are no differences between the results of these two queries:

```
SELECT LastName
FROM customer
WHERE LastName = 'Daly'
UNION ALL
SELECT LastName
FROM customer
WHERE LastName = 'Dixon'
ORDER BY LastName;

SELECT LastName
FROM customer
WHERE LastName = 'Daly'
    OR LastName = 'Dixon'
ORDER BY LastName;
```

TRY IT NOW

Execute the two previous queries to verify they both return the same results.

Both queries will return a result set with three rows: two for “Daly” and one for “Dixon”, although they do so in different ways. The first query executes two SELECT statements against the customer table and then combines the results, with each query searching for rows that meet a single condition. The second query instead executes a single SELECT, which searches for rows that meet multiple conditions.

TIP As you progress in your knowledge of SQL, it’s always good to know different ways to produce the same results, as there may be situations where one way may perform better than others. Depending on factors we will cover in later chapters, a query with a UNION ALL may produce results much faster than a similar one with an OR, even though the UNION ALL is executing two SQL statements to find the same results. For now, just remember to always consider different ways to find results if your query performs more poorly than expected.

One other difference between UNION and UNION ALL is that queries with UNION are generally slower than those with UNION ALL because the RDBMS has more work to do in removing the duplicate rows. However, if the result set from a UNION ALL is very large and full of duplicates, UNION queries could be faster, as there would be less data in the result set to send over the network. Keep this in mind when writing SQL statements that query hundreds of gigabytes of data or more, where using either UNION or UNION ALL will provide sufficient results.

And lastly, UNION ALL can be used to achieve the same results as a FULL OUTER JOIN, which we noted in the last chapter is not a supported type of join in MySQL, Maria DB, or SQLite.

10.4 Emulating FULL OUTER JOIN in MySQL

As noted at the end of Chapter 9, a FULL OUTER JOIN will return not only the rows that match between two tables, but also the unmatched rows from both tables. Figure 10.6 shows how we would represent these results in a Venn diagram, were we able to write such a query by searching for promotion codes shared by the promotion and orderheader tables.

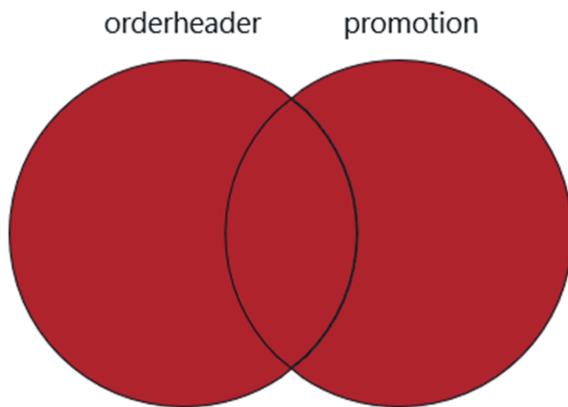


Figure 10.6 A Venn diagram of the values included in a query of the orderheader table with a full outer join of the promotion table

Here is what our SQL used to select this will look like. This type of query won't execute in MySQL, but it will execute on another RDBMS that allows the use of the FULL OUTER JOIN:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
FULL OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
```

The results of a FULL OUTER JOIN are very similar to executing a LEFT JOIN and RIGHT JOIN at the same time, but since we can't do this in MySQL, we can emulate it with the left and right joins combined with a UNION ALL. UNION ALL is preferred in this case because it does not remove duplicate rows, which may exist between the tables and would be returned with a FULL OUTER JOIN.

The one tricky thing to remember is that when we are emulating a FULL OUTER JOIN in this way, we need to modify one of the joins to exclude common values. If we don't do that, we will end up with duplicate rows, as both left and right joins include the common values that an INNER JOIN would return.

The following query demonstrates how we can emulate a FULL OUTER JOIN in MySQL using UNION ALL. We will prevent the duplicate representation of matching rows by excluding them from our second SELECT by filtering on WHERE oh.PromotionID IS NULL, which we learned about in the previous chapter:

```

SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
LEFT JOIN promotion p
    ON oh.PromotionID = p.PromotionID
UNION ALL
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
RIGHT JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.PromotionID IS NULL;

```

This query returns 53 rows, which include:

- Rows that match PromotionID values in both tables
- Rows in orderheader which do not contain PromotionID values
- Rows in promotion that have PromotionID values that were not used in the orderheader table

It may be helpful to use Venn diagrams to show us specifically what each of the two SELECT statements in our query is doing. The first query is finding the first two of our three sets of rows noted earlier, which are rows that match PromotionID values in both tables, and rows in orderheader which do not contain PromotionID values. This is represented by the diagram in figure 10.7.

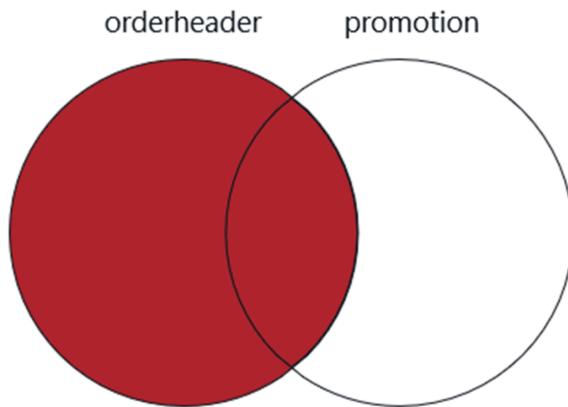


Figure 10.7 A Venn diagram of the values included in a query of the `orderheader` table with a left outer join of the `promotion` table

The second query is finding the last item, which are rows in `promotion` that have `PromotionID` values that were not used in the `orderheader` table. These can be graphically represented by the diagram in figure 10.8.

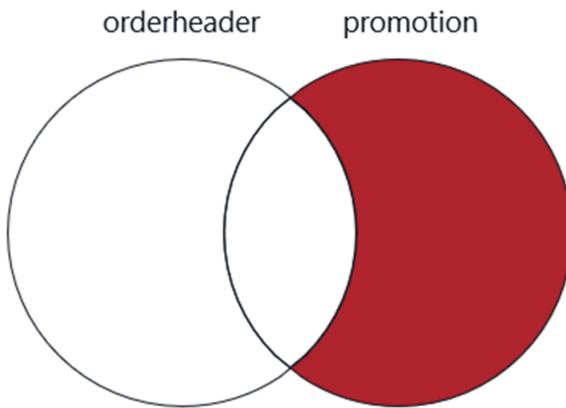


Figure 10.8 A Venn diagram of the values included only in a query of the `promotion` table, with a right outer join of the `orderheader` table

By using a `UNION ALL`, we have effectively combined all these results into a single result, as represented in the diagram in Figure 10.6.

`UNION` and `UNION ALL` are useful set operators, but for nearly every RDBMS there are two other set operators that you should know about: `INTERSECT` and `EXCEPT`.

WARNING MySQL did not support `INTERSECT` and `EXCEPT` until version 8.0.31. If you are using a version earlier than that, you will encounter errors when attempting to use these operators.

10.5 INTERSECT

Another set operator that may prove useful to know is INTERSECT, which can be used to return results similar to the results of a query with an INNER JOIN. However, there are two important differences between their results:

1. Where INNER JOIN will return duplicate values, INTERSECT will not. This is similar to the differences in the results of UNION ALL compared to UNION.
2. An INNER JOIN will never return null values, as nothing cannot equal nothing. Because INTERSECT is looking for common values between two sets of data and not evaluating equality, the results of INTERSECT will also include any null values that match in the results of two queries.

Let's review an inner join to demonstrate the differences in usage. Here's how we can use an INNER JOIN to find promotion ID values included in both the orderheader and promotion tables:

```
SELECT
    oh.PromotionID
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

If we were to write this query for an RDBMS that supports INTERSECT, our query would look like this:

```
SELECT
    PromotionID
FROM orderheader
INTERSECT
SELECT
    PromotionID
FROM promotion;
```

Although we used only one column in this example, INTERSECT does support the use of multiple columns in your SELECT statements. Although the column names do not need to be the same, all queries must have the same number of columns, and the columns must be selected in the same order for INTERSECT to be able to evaluate them.

10.6 EXCEPT

There is another set operator frequently included in other RDBMSs that can be used to return data in one set that is not included in a second set. That is the EXCEPT set operator, and it excludes results similar to a method we learned about with left joins in the last chapter.

NOTE The EXCEPT operator is not supported in Oracle; however, Oracle does have a MINUS operator that is identical in functionality and usage.

Suppose we wanted to find all the promotion ID values in the promotion table that were not used in any orders in the orderheader table, as represented by the diagram in figure 10.9. We already know we can find them with a statement like this:

```
SELECT
    p.PromotionID
FROM promotion p
LEFT JOIN orderheader oh
    ON p.PromotionID = oh.PromotionID
WHERE oh.PromotionID IS NULL
```

This kind of query is represented by the following diagram.

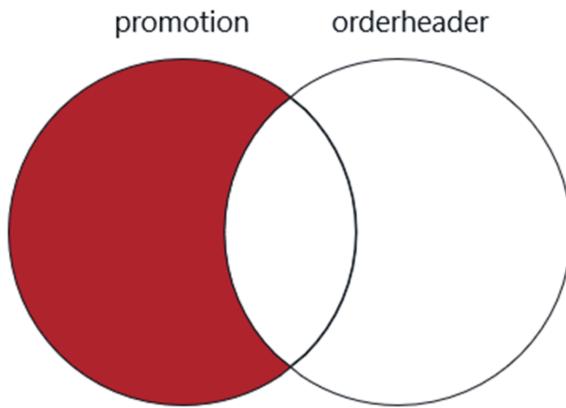


Figure 10.9 A Venn diagram of the values included only in a query of the promotion table, with a left outer join of the orderheader table

We can achieve a result set very similar to this use of a LEFT JOIN that filters on null matches in the joined table by using EXCEPT:

```

SELECT
    PromotionID
FROM promotion
EXCEPT
SELECT
    PromotionID
FROM orderheader;

```

As with INTERSECT, EXCEPT has two significant differences in results from LEFT JOIN:

1. Whereas a LEFT JOIN will return duplicate values, EXCEPT will not.
2. A LEFT JOIN will never return null values, as nothing cannot equal nothing.

Because EXCEPT is looking for common values between two sets of data and not evaluating equality, the results of EXCEPT will also include any null values that exist in the results of the first query and not the second.

Although INTERSECT and EXCEPT are used far less frequently than INNER JOIN and LEFT JOIN, keep them in mind if you are using a different RDBMS and you need to have null values returned in your result set.

That's plenty of information on how to use set operators. In the next chapter, we will examine other ways to join tables and other sets of data, using logical operators.

10.7 Lab

1. In this chapter we talked about how the column names in queries UNION and UNION ALL will come from the first SELECT statement. What do you think will happen with a query like this one that has no column name for the last column in the first query? Try it and find out:

```

SELECT FirstName, LastName, 'customer'
FROM customer
UNION
SELECT FirstName, LastName, 'author' TableName
FROM author
ORDER BY LastName, FirstName;

```

2. Considering there are rows in the customer table for Cora Daly and Kevin Daly, will the results of these two queries be the same? If not, what will be the differences?

```
SELECT LastName  
FROM customer  
WHERE FirstName = 'Cora'  
OR FirstName = 'Kevin';  
  
SELECT LastName  
FROM customer  
WHERE FirstName = 'Cora'  
UNION  
SELECT LastName  
FROM customer  
WHERE FirstName = 'Kevin';
```

3. We've looked at the different behaviors for UNION and UNION ALL, but we haven't used them both in the same query. It's inadvisable to try to use them both in the same query, and depending on the RDBMS you are using, you may get an error message when attempting to use them both in the same query. Even if you can, the results can be unpredictable. To demonstrate this, try executing these two similar queries and notice they have different results. Why do you think the results are different?

```

SELECT LastName
FROM customer
WHERE FirstName = 'Cora'
UNION
SELECT LastName
FROM customer
WHERE FirstName = 'Kevin'
UNION ALL
SELECT LastName
FROM customer
WHERE LastName = 'Daly';

SELECT LastName
FROM customer
WHERE LastName = 'Daly'
UNION ALL
SELECT LastName
FROM customer
WHERE FirstName = 'Kevin'
UNION
SELECT LastName
FROM customer
WHERE FirstName = 'Cora';

```

10.8 Lab answers

1. Since there is no column name supplied for the last column in the first SELECT statement, the literal value “customer” is used for the final column name in the result set.
2. The results will not be the same. The first query using OR for filtering will return two rows—one for each match. The second query with the UNION will return only one row because UNION removes duplicates from the results.
3. The first query returns three rows, while the second query returns one. At first glance, this may seem confusing since the second query simply reverses the order of the SELECT statements.

The answer lies in precedence, which indicates the order in which the SELECT statements are presented. The first UNION or UNION ALL is applied first, and then the second one is applied next. Remember, UNION eliminates duplicates, and UNION ALL does not.

In the first query, the first two SELECT statements each return one row with the value “Daly”, and since the UNION eliminates duplicates, the result is one row—so far. But the third SELECT statement returns two rows, which when evaluated with a UNION ALL, creates a result set of three rows—one row from the first two SELECT statements and two rows from the last SELECT statement.

In the second query, we combine the first SELECT statement, which returns two rows, with the second SELECT statement, which returns one row, and we do this with a UNION ALL. If we only executed this part of the query, we would get three rows returned. But we then combine another SELECT statement that returns one row with a UNION, which will then remove all the duplicates from our result set. This is why the second query returns only one row.

If this seems more than a little confusing, don't worry. As long as you avoid combining UNION and UNION ALL in the same query, you'll never have to worry about this headache.

11

Using subqueries and logical operators

In the previous chapter, we expanded the scope of our thinking a bit. We saw how we can use SQL to not only query tables but also, with the help of set operators such as UNION or INTERSECT, combine the results of two or more SELECT statements to form a single result set. In this chapter, we're going to build on that knowledge by examining an important method of evaluating the results of multiple SELECT statements in the same query: the subquery.

Subqueries are simply queries nested into another query. We use subqueries when we can't achieve the desired results from a single SELECT statement, so instead of writing two or more queries, we combine them into a single query. Don't worry, this isn't as complicated as it sounds.

By the end of this chapter, you'll see how subqueries will allow us to evaluate the results of SELECT statements in ways beyond the capabilities of the set operators we learned about in previous chapters. We have a lot of ways to use subqueries to discover, so let's get started.

11.1 A simple subquery

As we've noted throughout the book, a SELECT statement is a type of SQL query that returns a set of data known as a result set. So far, you've executed dozens of queries that produce result sets. You've queried one or more tables, which produced result sets that also look a lot like tables. By that, I mean the results have rows and columns, and the columns have names.

Let's take that a step further. Since query results produce a result set similar to a table, we can evaluate the results of SELECT statements in many of the same ways we can evaluate the data in a table. This means we can join and filter these results as if they were tables, and the way we do this is with subqueries.

Rather than continuing to speak theoretically, let's look at an example. Suppose we want to find the order ID and order date for any orders placed after a particular order from a customer named Margaret Montoya. We're using this customer because they only placed one order.

If we wanted to verbally declare this, we might say the following:

"I would like the order ID, customer ID, and order date from the customer table, but I only want the orders placed after the one order placed by the customer named Margaret Montoya."

This is the first time we've declared something in a compound sentence, with two separate clauses. Knowing what we've covered so far, you could write two separate queries to get the desired results. The first query, to find the order placed by Margaret Montoya, might look like this:

```
SELECT
    oh.OrderID,
    oh.OrderDate
FROM orderheader oh
INNER JOIN customer c
    ON oh.CustomerID = c.CustomerID
WHERE c.FirstName = 'Margaret'
    AND c.LastName = 'Montoya';
```

The result of this query shows Margaret Montoya placed only one order, on April 23, 2021. Now that you have that date, you could include it in a second query to find the desired results shown in figure 11.1, which might look something like this:

```
SELECT
    OrderID,
    CustomerID,
    OrderDate
FROM orderheader
WHERE OrderDate > '2021-04-23';
```

OrderID	CustomerID	OrderDate
1044	19	2021-06-06 00:00:00
1045	11	2021-10-01 00:00:00
1046	4	2021-11-13 00:00:00
1047	19	2021-11-28 00:00:00
1050	1	2022-03-10 00:00:00

Figure 11.1 The order ID, customer ID, and order date of all orders placed after Margaret Montoya placed her only order on April 23, 2021

Writing two queries to find this result is cumbersome, which is why we would replace the hard-coded date of April 23, 2021, with a subquery instead. All we need to do is replace our hard-coded order date of '2021-04-23' with the first query, which is now a subquery, and surround the subquery with parentheses.

WARNING When filtering with a subquery, we can only have one column returned. This is because SQL allows us to evaluate only one value or set of values at a time in the WHERE clause. If you select more than one column in the subquery used next, then your query will result in an error.

Here is what a SELECT statement with a subquery that produces the same results as the two previous queries might look like:

```
SELECT
    OrderID,
    CustomerID,
    OrderDate
FROM orderheader
WHERE OrderDate > (
    SELECT
        oh.OrderDate
    FROM orderheader oh
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.FirstName = 'Margaret'
        AND c.LastName = 'Montoya'
);
```

The results of this query are the same as the results shown in Figure 11.1, as we have now combined the logic of two queries into one.

The subquery just shown has the containing parenthesis on different lines above and below the subquery. Although you don't need to format your subqueries in this way, this method of formatting does have benefits. First, it's a bit easier to read than if you placed the parentheses right up against the beginning and end of the subquery. Perhaps more importantly, it's easier to drag the cursor over the subquery and highlight just the subquery, and execute it alone. This is helpful when writing your own SQL to verify that the subquery is producing the desired results.

TRY IT NOW

Execute the preceding subquery entirely, and then highlight and execute just the lines of the subquery inside the parentheses to validate that the value returned is April 23, 2021.

In this query we used a comparative operator, `>`, with our subquery, but other comparison operators such as `=` and `<>` have the limitation of being able to compare only one value. To unlock the full potential of subqueries, we will need to use an entirely different set of keywords, known as *logical operators*.

11.2 Logical operators and subqueries

Logical operators are a bit like comparison operators in that they test for whether some condition is true or false or unknown. You've already used a few of these: `IN`, `NOT IN`, `BETWEEN`, and `LIKE`.

Some comparison operators are unable to evaluate a result set with more than one row. For example, let's look at a subquery that returns more than one row.

Order ID 1034 is an order that includes more than one title, as we can see in figure 11.2. If we need to write SQL to find what titles are included, we can write something like this:

```
SELECT
    t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON oi.TitleID = t.TitleID
WHERE OrderID = 1034;
```

	TitleName
▶	The Join Luck Club
	Catcher in the Try
	Anne of Fact Tables
	The DateTime Machine

Figure 11.2 The four titles included in order 1034

Let's rewrite this query using a subquery by moving the part of the query that filters on order ID into its own query, place that as a subquery in the `WHERE` clause, and filter on where the title ID from the title table is equal, using `=` to match the value of the results of our subquery:

```

SELECT
    t.TitleName
FROM title t
WHERE TitleID = (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
);

```

But this query doesn't work. If you try executing it, you will see in the Output panel an error indicating the "Subquery returns more than 1 row." This is true, as we know from the previous query this order includes four titles, which means four rows are returned by the subquery. Our subquery can't be evaluated by = because that comparison operator is trying to determine whether every title ID in the title table equals a single value. This query would work if there were only one title in the order, but there are four, and unfortunately the = operator can't evaluate more than one value.

TRY IT NOW

Execute the preceding query and notice the error in the Output panel.

This is where our first logical operator, ANY, can help.

11.2.1 The ANY and IN operators

The ANY logical operator will evaluate a set of values to see if any of them have equality to the values we are attempting to match—hence, the name ANY. You can think of it like a helper to = (or any other comparison operator), allowing any value included in the subquery.

We can get the previous query to work by simply adding the ANY operator after = in the WHERE clause:

```

SELECT
    t.TitleName
FROM title t
WHERE TitleID = ANY (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
);

```

The results of this query are the same as the results shown in Figure 11.2.

We can verbally declare what we are doing, like this:

"I would like the title name from the title table, and I would like the titles to match any of the titles from order 1034."

NOTE Most RDBMS, including MySQL, also support the SOME logical operator, which is identical to ANY in usage and function. However, it is used much less frequently than ANY.

We can also get these same results using a different logical operator than we have used before: the IN keyword. We can replace = ANY with IN and get the same results. This can be verbally declared in a similar way:

"I would like the title name from the title table, and I would like the titles to be in titles from order 1034."

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID IN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
);
```

So now that you have two logical operators you can use with a subquery that gets the same results, which should you choose? Well, that depends on if you need to also use a comparison operator. If you need to use `>`, `>=`, `<`, or `<=`, then you would need to use ANY because IN doesn't allow for those kinds of comparisons. However, if you do not need one of those comparison operators, you will see IN used for these kinds of subqueries far more often than = ANY is used.

Now let's look at the opposite way to filter, on excluding the results of our subquery.

11.2.2 The ALL and NOT IN operators

Suppose we need to find the names of the titles that are not in order 1034, as shown in figure 11.3.

"I would like the title name from the title table, and I would like the titles to not be in titles from order 1034."

Just as we added the word "not" to our verbal declaration, we can do the same thing with our SQL:

```

SELECT
    t.TitleName
FROM title t
WHERE TitleID NOT IN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034);

```

TitleName
Pride and Predicates
The Great GroupBy
The Call of the White
The Sum Also Rises

Figure 11.3 The four titles not included in order 1034

There are eight total titles in our sqlnovel database. Order 1034 includes four of them, as shown in Figure 11.2, so now we know the other four titles that were not in that order. What our SQL statement is doing now is evaluating all the titles in order 1034 and then finding the titles in the title table that aren't any of those included in the order.

Which brings us to another way we might be tempted to verbally declare the desired results of this query:

"I would like the title name from the title table, and I would like the titles to not match any of the titles from order 1034."

At first glance, we might think we can use the ANY operator to also get these results by using it with the not equals operator, <>:

```

SELECT
    t.TitleName
FROM title t
WHERE TitleID <> ANY (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
);

```

Although this query will execute, it won't provide the desired results. What this query is doing is evaluating all the titles in the title table and seeing which ones do not match any of the titles in our subquery. Because our subquery contains more than one title, every title in the title table will be a match, as there is at least one title in the subquery that isn't the same.

TRY IT NOW

Execute this query, and notice that it returns every title in the title table.

To get the results we want, we need to use the ALL operator instead of ANY because what we really want is titles that don't match all of the titles in the subquery.

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID <> ALL (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
);
```

Executing this query will provide the same results as shown in Figure 11.3, which is what we intended. As with IN and ANY, the decision to use NOT IN or ALL will come down to whether a comparative operator is required.

We haven't mentioned it yet, but be aware that by using subqueries, we are asking the RDBMS to effectively execute two queries at once and evaluate the results of one against the other. Generally speaking, this requires more processing and memory to complete our queries, so when writing SQL, you should carefully consider whether the use of a subquery is absolutely necessary.

That said, two other operators make for more efficient subqueries than the ones we've covered so far: EXISTS and NOT EXISTS. What makes these operators appealing is that when used, they don't evaluate the values of every row in the subquery, but rather they only check to see if there are *any* matching rows. Once a match for a value is found, other matches are not evaluated for equality or inequality.

11.2.3 The EXISTS and NOT EXISTS operators

We will first look at EXISTS, which is used similarly to our use of = ANY and IN in this chapter to find the titles included in order 1034. The difference is that, when using EXISTS, we will need to include a kind of join in the WHERE clause of the subquery.

Here is what this would look like:

```

SELECT
    t.TitleName
FROM title t
WHERE EXISTS (
    SELECT TitleID
    FROM orderitem oi
    WHERE OrderID = 1034
    AND t.TitleID = oi.TitleID
);

```

Executing this query will provide the same results as shown in Figure 11.2, which returns the names of all the titles in order 1034. Notice that we have not only used EXISTS in the WHERE clause, but we now have an additional line in the WHERE clause of the subquery that says AND t.TitleID = oi.TitleID. This join is where the evaluation in the subquery takes place. The evaluation is no longer from the value we are selecting, so what we actually put in the SELECT clause of our subquery no longer matters.

For this reason, you will often see subqueries used with EXISTS to have something that seems nonsensical in the SELECT clause, like this query which has SELECT 1 in the subquery:

```

SELECT
    t.TitleName
FROM title t
WHERE EXISTS (
    SELECT 1
    FROM orderitem oi
    WHERE OrderID = 1034
    AND t.TitleID = oi.TitleID
);

```

TRY IT NOW

Execute the preceding query and see for yourself how it still returns the same results as shown in Figure 11.2. Try replacing the 1 in the SELECT clause of the subquery with any other value to see how the value there is now irrelevant.

Conversely, we can use NOT EXISTS in the same way, to find the titles that are not included in order 1034. Executing the following query will return the same results shown in Figure 11.3:

```

SELECT
    t.TitleName
FROM title t
WHERE NOT EXISTS (
    SELECT 1
    FROM orderitem oi
    WHERE OrderID = 1034
        AND t.TitleID = oi.TitleID
);

```

Again, the main reason to use EXISTS or NOT EXISTS with subqueries is when we are querying larger sets of data, as they can provide better performance than using IN/NOT IN, ANY, or ALL.

11.3 Subqueries in other parts of a query

So far in this chapter we have only looked at subqueries used for filtering in the WHERE clause, but we can also use subqueries in other clauses as well. For instance, we can write a query to return the same results as those shown in Figure 11.2 with a join in the FROM clause.

11.3.1 Subqueries in the FROM clause

To do this, we simply move our subquery into a join in the FROM clause—in this case, an inner join. We won’t need any operators to do this, since we aren’t evaluating the subquery for filtering. We are simply joining the results of the subquery, and any evaluation is occurring with the ON part of the join.

Also, because our subquery doesn’t have an actual name, we’ll also need to use an alias so that we can join it to another table:

```

SELECT
    t.TitleName
FROM title t
INNER JOIN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
) oisq
ON t.TitleID = oisq.TitleID;

```

By moving the subquery into the FROM clause, we're now treating our subquery results as if they were a table, with those results being joined to the title table by TitleID. The results of the subquery aren't actually a table, but they will have to be computed first by the RDBMS before we can determine how many possible rows from the results in our subquery can be joined to the title table.

What's interesting here is that because the subquery is in the FROM clause, it can be used in our query like a table would be. As such, we are no longer limited to having one column in our subquery, so we can add more columns to the subquery if needed for joining or filtering purposes.

TRY IT NOW

Execute the preceding query, and then change SELECT TitleID to SELECT TitleId, OrderID and execute that as well.

We can also use a join in the FROM clause to find values that are not in the subquery, using the LEFT OUTER JOIN method we used in Chapter 9. If case you don't recall, we can use this kind of join with a filter on the null values in the second table, or in this case the subquery, being joined to find values existing in the first table but not in the joined table.

```
SELECT
    t.TitleName
FROM title t
LEFT JOIN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
) oisq
    ON t.TitleID = oisq.TitleID
WHERE oisq.TitleID IS NULL;
```

The results of this query will be the same as those shown in Figure 11.3.

11.3.2 Subqueries in the SELECT clause

A final way to use subqueries is in the SELECT clause. We can also get the same results of title names as those shown in Figure 11.2 by using a subquery in the SELECT clause, although we have to completely rearrange the query by switching the subquery from the filtering query on orderitem to the selection of the title name from title:

```

SELECT
    (SELECT TitleName from title t where t.TitleID = oi.TitleID) as TitleName
FROM orderitem oi
WHERE oi.OrderID = 1034;

```

WARNING This is a highly unusual way to find this result set. This example is only being shown so that you can see what a subquery in the SELECT clause looks like. Writing subqueries in the SELECT clause is rarely the best way to write SQL.

All right, we have covered several different ways to use subqueries and to see the options they afford us in writing SQL. It should be noted however that subqueries should be used sparingly, as they can often have a negative effect on query performance. The fact that each subquery will execute an additional SELECT statement means our SQL statements with subqueries typically create more work for the RDBMS.

In the next chapter, we will look at how to group sets of data to find calculations like the minimum and maximum values of those sets. You will even get to put your new subquery skills to use in the lab.

11.4 Lab

1. Write a query using a subquery with IN to get the names of the title(s) in the only order placed by Joe Pagenaud.
2. Take a look again at the queries used in section 11.1, where we tried to find the orders placed after Margaret Montoya's order. Write a similar query to find the order ID, customer ID, and order date from any orders placed after all of Cora Daly's orders.
3. Selecting 1 divided by 0 returns a null value. Could we use SELECT 1/0 instead of SELECT TitleID in the SELECT clause of the subquery of the first query in 11.2.3 and still get the correct results?

11.5 Lab answers

1. There are many ways to do this, depending on which queries you join in the subquery. Here is one way.

```

SELECT
    t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
WHERE oi.OrderID IN (
    SELECT
        oh.OrderID
    FROM orderheader oh
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.FirstName = 'Joe'
        AND c.LastName = 'Pagenaud'
);

```

2. Your query may vary, but here is a way to find the intended order information.

```

SELECT
    OrderID,
    CustomerID,
    OrderDate
FROM orderheader
WHERE OrderDate > ALL (
    SELECT
        oh.OrderDate
    FROM orderheader oh
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.FirstName = 'Cora'
        AND c.LastName = 'Daly'
);

```

3. Yes, because the value or column used in the SELECT clause of a subquery is not evaluated by EXISTS or NOT EXISTS.

12

Aggregating data

If you're accustomed to working with spreadsheets and not relational data in a database, the last three chapters may have been a bit challenging for you. After all, in spreadsheets you often work with just a single set of data instead of multiple sets. So, if you find the concepts in those chapters completely new to you, then take heart because in this chapter we are going to cover concepts that should be very familiar to most spreadsheet users.

One of the useful aspects of spreadsheets is that they allow us to quickly do mathematical calculations on a range of data. For example, if we want to find the sum total of all values in a column, we can click the AutoSum button, which will place the desired sum amount in a particular cell. If you highlight that cell, you'll see that the spreadsheet has the used word SUM with the defined range of cells. That word, SUM, represents something called a *function*, which is a command that performs a predefined calculation.

Although we have no button in the SQL language to automatically calculate sum totals, we do have functions like SUM to help us perform mathematical calculations. Moreover, in a relational database we have much more flexibility in the way we can perform these calculations than we have in spreadsheets.

Let's look at a particular group of these functions, known as aggregate functions.

12.1 Aggregate functions

Throughout the rest of this book, we will be discussing different kinds of functions in SQL. A *function* is a keyword that allows us to easily perform calculations or other actions. There are many functions in SQL for calculating all sorts of values for converting dates, formatting data, and so much more.

In this chapter, we are focusing on the main *aggregate* functions, which are functions that perform a calculation over a range of data in a column. If you need to perform basic calculations, these aggregate functions will be indispensable.

12.1.1 The SUM function

The most basic function we can use is the SUM function, which will return the sum total of values for a column of data. For example, if we wanted to know the total number of titles ordered, we could sum the Quantity column in the orderitem table. We could verbally declare this, using the word “sum” to describe what we want.

“I would like the sum of the quantity of titles in the orderitem table.”

This isn’t too far off from our SQL, which would look like this, with the results shown in figure 12.1:

```
SELECT SUM(Quantity)
FROM orderitem;
```

SUM(Quantity)
70

Figure 12.1 The quantity of all orders is shown in a column with no alias. The default column name is the calculation.

There are a few things to note before we go any further. First, we need to use parentheses to identify what column we are choosing for the sum. If we don’t use the parentheses, the query will result in a syntax error.

Also, if you execute this query, you will notice the column will have our calculation as the name of the column, which may or may not be helpful. When using aggregate functions, we usually want to indicate a name for our column by using an alias. This helps us identify the meaning of our returned values. If we were to execute this query again, we should modify it to use an alias that reflects the aggregate calculation:

```
SELECT SUM(Quantity) as TotalQuantity
FROM orderitem;
```

Keep in mind the SUM function is intended to be used only with numeric data values. If you try to use this function with date or character values, the result may not be meaningful.

TIP When using an alias for an output column, avoid using the name of the table column for the alias. In addition to possibly creating confusion for anyone who might read your output, some RDBMSs will not allow this.

While SUM adds the combined total of all values together, if we want to know the quantity of values that exist in each column, we’ll need to use a different function.

12.1.2 The COUNT function

The COUNT function counts the number of rows in a column. This seems relatively obvious, but it's here we should note one important feature of all aggregate functions: by default, they exclude null values.

Let's look at the orderheader table and see how many rows there are. We can do this easily by selecting the COUNT of all the order IDs, of which every row has a value, and see the result in figure 12.2:

```
SELECT COUNT(OrderID) as TotalOrders
FROM orderheader;
```

TotalOrders
50

Figure 12.2 There are 50 rows in the orderheader table with a value for OrderID, which is all the rows in the table.

The results indicate we have 50 rows in the orderheader table, which is correct. However, if we try to count the number of promotion codes used by selecting the COUNT of the PromotionID column, we're going to get a different result as shown in figure 12.3:

```
SELECT COUNT(PromotionID) as TotalOrdersWithPromotionCode
FROM orderheader;
```

TotalOrdersWithPromotionCode
20

Figure 12.3 There are only 20 rows in the orderheader table that have a value for PromotionID.

The results now show only 20 rows, which means only 20 of the 50 rows in the orderheader table have a value for PromotionID. The other 30 rows have null for PromotionID.

The COUNT function also has a unique and widely used feature, which is the ability to return the number of rows in a table without specifying a column. If we didn't know the names of any columns in the orderheader table, we could easily determine this by using the asterisk (*) we learned about when selecting all columns back in Chapter 3:

```
SELECT COUNT(*) as TotalOrders
FROM orderheader;
```

The results of this query will be the same as those in figure 12.2. This is noteworthy because even if there are null values in any of the columns, selecting COUNT(*) will always return the total number of rows in a table.

WARNING SELECT COUNT(*) is a useful method for quickly determining the number of rows in most tables, but be careful when using this SQL with tables that have millions or billions (or more) rows. This kind of query can use excessive computer resources and cause delays for other queries.

12.1.3 The MIN function

The MIN function returns the minimum, or lowest, non-null value for a column. For example, if we want to find the least expensive item from all orders, as shown in figure 12.4, we can use a query like this:

```
SELECT MIN(ItemPrice) as MinimumItemPrice
FROM orderitem;
```

MinimumItemPrice
4.95

Figure 12.4 The minimum price of any item in the orderitem table is \$4.95.

The MIN has a commonly used partner function: the MAX function.

12.1.4 The MAX function

While the MIN function returns the lowest value for a row, the MAX function returns the maximum, or highest, value. Let's change the function used in the previous query to find the most expensive price for any item, as shown in figure 12.5:

```
SELECT MAX(ItemPrice) as MaximumItemPrice
FROM orderitem;
```

MaximumItemPrice
12.95

Figure 12.5 The maximum price of any item in the orderitem table is \$12.95.

While the SUM function can be used only with numeric data, in MySQL and many other RDBMSs you can use the MIN and MAX functions with non-numeric data. When used with non-numeric data, the functions will return the first or last values respectively, as if the data was sorted in ascending order on that column. Although for dates this is rarely problematic, the warnings in previous chapters about collation would apply here as well, as sometimes lower- and uppercase letters, as well as non-alphabetic characters, are ranked differently by different collations.

TRY IT NOW

Write a short query to select the MIN value for the FirstName column in the author table.

12.1.5 The AVG function

The last aggregate function we're going to use is AVG, which will compute the average of all non-null values in a column. If we wanted to find the average price of the titles in the title table, as shown in figure 12.6, we could use the AVG function like this:

```
SELECT AVG(Price) as AveragePrice
FROM title;
```

AveragePrice
9.700000

Figure 12.6 The average price of all titles in the title table

It looks like the average price is about \$9.70 for the titles in our database, and you surely have noticed a lot of extra zeroes in the result. This is because the AVG function is attempting to calculate the average value to a higher level of precision. It's not really a problem because it is an accurate value, and in Chapter 14 we will discuss how to modify this if needed.

Like the SUM function, the AVG function should be used only with numeric values.

12.1.6 Filtering and aggregating combined values

We haven't tried it yet, but we can also use a filter with our aggregate functions. Let's say we wanted to determine the average price of all titles published since January 1 of 2018. Let's try our verbal declaration for this.

"I would like the average price of all titles with a publication date greater than January 1, 2019."

This easily converts to SQL query:

```
SELECT AVG(Price) as AveragePrice
FROM title
WHERE PublicationDate > '2019-01-01';
```

We can even combine the use of our functions in the same query, such as if we wanted to know the dates of the first and last order dates from the orderheader table. We can use the MIN and MAX values since they work with date values:

```
SELECT
    MIN(OrderDate) as FirstOrder,
    MAX(OrderDate) as LastOrder
FROM orderheader;
```

Something else we can do with aggregate queries is use a mathematical calculation inside the parentheses. We need to do this in queries that require the values of more than one column, such as determining the total dollar value of all items sold. If we conclude that the total dollar value for any row in items is determined by multiplying the Quantity by the ItemPrice, we can use that calculation with a SUM function to determine the total sales value for all the rows in our database, with the result shown in figure 12.7:

```
SELECT SUM(Quantity * ItemPrice) as TotalOrderValue
FROM orderitem;
```

TotalOrderValue
573.50

Figure 12.7 The total value of all orders in the orderitem table is \$573.50.

As easy as this is to determine the total overall sales, until now we have been limited to evaluations of values at the table level, or evaluations based on filtered data from a table. What if we wanted to know the SUM of each order? Or the quantity of items sold for each promotion code? Or the number of titles sold by each author?

If we want to analyze data at a deeper level, we need a new set of keywords: GROUP BY.

12.2 Aggregating data with GROUP BY

GROUP BY isn't just a couple of keywords. It's an entirely new clause, and it allows us to divide one set of data into groups on which we can perform our aggregations. That explanation may seem theoretical, so let's look at a practical example.

12.2.1 GROUP BY requirements

In the previous query, we selected the total dollar value for all orders. If we wanted to find the total dollar value for each individual order, we would need to divide our data into groups of values for each order and then perform the same calculation as we did previously. We divide our values by grouping them, which in this case is by order ID. Given the formula we have used previously, a verbal declaration might look something like this:

"I would like the sum of the quantity multiplied by the item price of all the ordered items, and I want to group the sum by order ID."

Here's how this looks in a query. We'll add an ORDER BY to sort the data by order ID for the readability of the result shown in figure 12.8:

```
SELECT OrderID, SUM(Quantity * ItemPrice) as OrderTotal
FROM orderitem
GROUP BY OrderID
ORDER BY OrderID;
```

OrderID	OrderTotal
1001	9.95
1002	9.95
1003	9.95
1004	9.95
1005	9.95
1006	7.95
1007	15.90
1008	7.95

Figure 12.8 These are 8 of the 50 rows returned showing the Order Total for all orders.

This is similar to the last query, except we are now grouping our data into logical sets of values for each order ID and then performing the calculation of SUM(Quantity * ItemPrice) on each set. This is done by first adding the GROUP BY OrderID and then adding ORDERID to our SELECT clause.

NOTE When using GROUP BY, every column in SELECT must either be included in the GROUP BY clause or have an aggregate calculation. If there is a column in SELECT that does not meet either of these requirements, you will get a syntax error.

You may have noticed that the GROUP BY clause is used after the FROM clause and before the ORDER BY clause. In fact, GROUP BY needs to be after the WHERE clause as well, if we were to have one. Knowing this, if we wanted to limit our orders to only those placed after January 1 of 2019, we will need to join to our orderheader table to add this, and we should probably alias the column names as well:

```

SELECT
    oi.OrderID,
    SUM(oi.Quantity * oi.ItemPrice) as OrderTotal
FROM orderitem oi
INNER JOIN orderheader oh
    ON oi.OrderID = oh.OrderID
WHERE oh.OrderDate > '2019-01-01'
GROUP BY oi.OrderID
ORDER BY oi.OrderID;

```

Adding this filter on OrderDate will reduce the result set from 50 rows to 21, but keep in mind we are still performing the calculation on each set of data based on order ID for those 21 order groups.

12.2.2 GROUP BY and null values

Another feature of the GROUP BY clause is that we can now perform aggregations on columns with null values. Do you remember from section 12.1.2 that we used the COUNT function to find promotions used in orders, and the null values were excluded? With the GROUP BY clause, null can now be accounted for.

Suppose we wanted to account for those 30 orders that were placed without a promotion code, as in figure 12.9. These would be orders that were not placed for a discounted price, and they would be represented by having a null value in the PromotionID column of the orderheader table. Because these are null values, these were excluded from our previous query with COUNT in section 12.1.2, but we can use GROUP BY to group the orders logically by promotion ID, since GROUP BY will include null values:

```

SELECT PromotionID, COUNT(*) as RowCount
FROM orderheader
GROUP BY PromotionID
ORDER BY PromotionID;

```

PromotionID	RowCount
NULL	30
1	3
2	4
3	3
4	1
5	2
6	1
7	2
9	2
12	2

Figure 12.9 These are the count of all PromotionID values in the orderheader table. This includes null values since GROUP BY does not exclude nulls.

In this query, we are filtering on a column included in the GROUP BY clause: PromotionID. But filtering can be a bit tricky when using GROUP BY because the aggregations cannot be filtered in the WHERE clause. For instance, if we wanted to find promotion codes that were used only a few times, then we would have to introduce another clause.

12.3 Filtering with HAVING

The HAVING clause is the partner clause to GROUP BY in that it is where we will filter on the aggregations we are calculating. It's used similarly to the WHERE clause, with the main difference that the WHERE clause filters rows, while the HAVING clause filters groups of aggregated values. All the methods for filtering you've already learned can be applied in the HAVING clause if needed.

Let's build an example. First, suppose we want to find the Promotion codes used in orders and to see which were used at least three times. We could start by writing a query to find all promotion codes used, joining the orderheader table to promotion on PromotionID, grouping the values by promotion code, and then counting the times a Promotion ID was used in the orderheader table. We will alias the tables and order the results for readability:

```

SELECT
    p.PromotionCode,
    COUNT(oh.PromotionID) as OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
GROUP BY p.PromotionCode
ORDER BY p.PromotionCode;

```

NOTE If you have been executing all the queries in this chapter, you may be wondering what happened to the null values in the results of the previous query. Well, those null values for PromotionID exist in the orderheader table, but they don't exist in the promotion table. And even if they did, an INNER JOIN would exclude them, so our results will only include values that match from both tables.

Now that we have our basic query to view which promotion codes were used and how often they were included in an order, we can add the HAVING clause to filter on codes used three or more times, with the results shown in figure 12.10:

```
SELECT
    p.PromotionCode,
    COUNT(oh.PromotionID) as OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
GROUP BY p.PromotionCode
HAVING COUNT(oh.PromotionID) >= 3
ORDER BY p.PromotionCode;
```

PromotionCode	OrdersWithPromotionCode
2OFF2015	3
2OFF2016	4
2OFF2017	3

Figure 12.10 The only promotion codes that have been used at least three times

Now that we've added our HAVING clause, we have filtered our results down to three rows. If we wanted to, we could use the alias of our aggregation in the HAVING clause, like this:

```
SELECT
    p.PromotionCode as PromoCode,
    COUNT(oh.PromotionID) as OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
GROUP BY p.PromotionCode
HAVING OrdersWithPromotionCode >= 3
ORDER BY p.PromotionCode;
```

The query should return the same results as those shown in figure 12.10. This may seem contradictory, since in earlier chapters we noted how we can't use a column alias in the WHERE clause. If this seems a bit confusing, now is probably a good time to talk about something important in SQL and the queries we write, which is the logical order in which the RDBMS reads our queries.

12.4 Logical query processing

Including this chapter, you've now learned about several clauses in your SQL statements and the order that you must write them. At a simplified level, SQL clauses are ordered like this:

1. SELECT
2. FROM (including JOINs)
3. WHERE (including ANDs and ORs)
4. GROUP BY
5. HAVING
6. ORDER BY

However, this isn't the order that the MySQL RDBMS is going to read your queries. It's actually reading them in this order:

1. FROM (including JOINs)
2. WHERE (including ANDs and ORs)
3. SELECT
4. GROUP BY
5. HAVING
6. ORDER BY

This order of how the RDBMS reads your queries is known as *logical query processing*, which defines the logical order in which the RDBMS is going to process your query. This is important to understand because it will not only help you troubleshoot your queries when they give unexpected or incorrect results but also help determine when you can use table and column aliases.

The order of logical query processing may seem strange, but it is an optimal order for the RDBMS to process your query.

1. Evaluate the data in the tables your query will use in the FROM clause.
2. Filter the data to reduce the result set in the WHERE clause.
3. Gather the columns to be returned in the SELECT clause.
4. Group those columns in the GROUP BY clause for aggregation.
5. Filter the aggregations in the HAVING clause.
6. Sort the results in the ORDER BY clause.

Now you can see why table aliases can be used throughout the query, since they are logically established in the earliest processing of our query in the FROM clause. You can also see why we can use column aliases established in the SELECT clause in the HAVING and ORDER BY clauses, but not in the WHERE clause.

WARNING Although this is the logical query processing order used by MySQL, other RDBMSs may have a different order where the SELECT clause is logically processed after GROUP BY and HAVING. For this reason, you probably shouldn't get in the habit of using column aliases in your HAVING clause.

12.5 The DISTINCT keyword

There is one more keyword to cover in our chapter on aggregations, and that is DISTINCT. It's a helpful keyword that is frequently used, but it also seems to be a bit misunderstood.

We can use the DISTINCT keyword in a SELECT clause to avoid having repeating values. For instance, if we wanted to see the names of all titles ever ordered as in figure 12.11, we could write a query like this using DISTINCT:

```
SELECT DISTINCT t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
ORDER BY t.TitleName;
```

TitleName
Anne of Fact Tables
Catcher in the Try
Pride and Predicates
The Call of the White
The DateTime Machine
The Great GroupBy
The Join Luck Club
The Sum Also Rises

Figure 12.11 The distinct title names that have been included in orders in the orderitem table

Although there are 50 orders in our table, with some orders placed for more than one title, our query only returns one row for each of the titles. DISTINCT can be very useful in quickly determining the range of values in any table, and I'm sure you'll see it used often in other people's queries.

So why is it noted in this chapter on aggregations? Because when you use SELECT DISTINCT, your RDBMS is basically doing an aggregation to return your distinct values, and that aggregation is extra work. By using DISTINCT in the previous query, we are basically asking the RDBMS to process this:

```
SELECT t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
GROUP BY t.TitleName
ORDER BY t.TitleName;
```

TRY IT NOW

Run the two previous queries with DISTINCT and GROUP BY respectively and notice how they have the same results as shown in figure 12.11.

Now that you are aware of what DISTINCT actually does, try to limit its usage in your SQL, especially with large data sets. Aggregating data isn't problematic when querying the small tables in our database, but it can be when querying larger tables elsewhere.

TIP One of the most common misuses of DISTINCT is when attempting to eliminate duplicates from query results when multiple data sets are joined. If you find yourself tempted to use DISTINCT to remove duplicates from the results of a query with multiple joins, be sure to take a second look at the join conditions to make sure you are joining on the correct columns. Incorrect joins are often the cause of unwanted duplicate rows in a result set.

12.6 Lab

1. Earlier in the chapter, we noted you can't use certain functions with certain data types. To better understand this, select the SUM of the OrderDate in the orderheader and see what the result is.
2. Why won't this query work?

```
SELECT
    p.PromotionCode as PromoCode,
    COUNT(oh.PromotionID) as OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE PromoCode = '2OFF2015'
GROUP BY p.PromotionCode
HAVING OrdersWithPromotionCode >= 3
ORDER BY p.PromotionCode;
```

3. Write a query to count the number of rows in the author table.
4. Write a query to select the minimum and maximum values of the publication dates from the titles table.
5. Earlier in the chapter, we determined the total dollar value of all orders using the equation Quantity * ItemPrice from the orderitem table. Write a query using GROUP BY to determine the average total dollar value for each individual order.
Hint: you may need to use a subquery.

12.7 Lab answers

1. You may have missed it earlier in the book, but it was noted how date and time values are actually stored as numeric values that your RDBMS can interpret as dates and times. Although it's practically useless to us, the value you see here is the RDBMS trying to make sense of using a SUM of date values.
2. The query will fail because a column alias is used in the WHERE clause, and the logical query processing order is to evaluate the WHERE clause before the SELECT clause. Because of this, the RDBMS doesn't know what "PromoCode" is when evaluating the WHERE clause, since the alias is determined later in the query in the SELECT clause.
3. To count the number of rows in a table, you can use COUNT(*)

```
SELECT COUNT(*)
FROM author;
```

4. To select the minimum and maximum publication dates, you can use the MIN and MAX functions:

```
SELECT
    MIN(PublicationDate) as FirstPublication,
    MAX(PublicationDate) as LastPublication
FROM title;
```

5. There are a few different ways you can do this, but the first is to group the total value of all orders as we did in section 12.2.1 and then select an average value of all those order totals, like this:

```
SELECT AVG(OrderTotals.OrderTotal)
FROM (
    SELECT OrderID, SUM(Quantity * ItemPrice) AS OrderTotal
    FROM orderitem
    GROUP BY OrderID
) OrderTotals;
```

13

Using variables

We've written and executed a lot of SQL queries so far, and a good number of those queries involved filtering the results on specific values. Through many examples, you have seen how to filter on a particular order or title ID or customer name or date range, and every time we have specified the literal value for filtering in our SQL. A *literal* value is something specific, such as the number 4 or the date "2020-10-06". Using literal values is helpful for learning and practice, but when you go to use SQL outside of this book, you'll need to be able to write queries that are more flexible.

For example, if you want to look at the total sales of a title for a given month, such as March 2021, you can write a query to do that now. However, what if you want to run a similar query in April? Or what if you need the total sales for a different title or a different range of dates? Does this mean you need to write a different query for each title and date range?

I assure you, you do not. All you need to do is learn how to use variables. A *variable* is a memory-based object that stores a value that, once defined, can be used repeatedly throughout a query or in subsequent queries. More importantly, the value that is stored can vary from one execution to another, which means the value is variable – hence the name.

Considering the flexibility that variables provide, you will most certainly be using them with great frequency throughout your SQL. Let's get started!

13.1 User-defined variables

Although there are a few different kinds of variables, the kind we will be using in this chapter is known specifically as *user-defined variables*. The name is self-explanatory, as the user (you or I) will be defining these variables, which means we will be assigning them a name and a value. These variables will all start with the at-sign (@), so when you see @ used in SQL, you will likely be looking at a variable.

Before we use any variable, we must declare it. Since chapter 2, we've been verbally declaring our intentions in English to help us understand the syntax of queries, but sometimes we also need to declare things in our SQL as well. Let's look at how we do this in MySQL.

13.1.1 Declaring your first user-defined variable

Declaring a variable typically requires two pieces of information to start: the *name* of the variable and the *value* of the variable. Let's suppose we want to write a query that will be filtering on title name. We could start with a sensible variable name of @TitleName and we could start with the value of "The Sum Also Rises". This could be verbally declared in a straightforward way.

"I would like to declare a variable named @TitleName, and I would like to assign it the value of 'The Sum Also Rises'."

The SQL used for this declaration will be similar in logic, using a new keyword: SET:

```
SET @TitleName = 'The Sum Also Rises';
```

As with many things in SQL, the syntax is similar to the order of our verbal declaration, where we declare a variable by setting its name with SET and then assign a value using = and a literal value.

NOTE When using SET, either = or := can be used as the assignment operator. Which you use is a matter of personal preference, although as you'll see later in the chapter, there is at least one instance where you must use := for your variable declaration.

One thing you may also have noticed is we didn't specify what data type we would use. That's because MySQL will determine the data type based on the value you used. In our example, we used a character string ('The Sum Also Rises') for the value of the variable, so our variable is a string data type.

Other permissible data types for variables include integer, decimal, and float, which are all numeric types of data. If the data for the variable doesn't fit one of the permissible types, then the RDBMS will convert the values to a permissible data type. Date and time values, which we have used throughout this book, will be treated as strings.

WARNING This method of declaring variables in MySQL is not universal. When using a different RDBMS such as SQL Server or PostgreSQL, you will have to first declare a user-defined variable using the DECLARE keyword and then assign it a specified data type.

If needed, we can confirm the value of our variable at any time with a simple SELECT statement, as shown in the results in figure 13.1:

```
SELECT @TitleName;
```

	@TitleName
▶	The Sum Also Rises

Figure 13.1 The results of selecting @TitleName, which shows the value for the variable. It also shows the variable name in the header.

Although selecting a variable to confirm its value may seem trivial, this is actually a method you will use quite a bit when using variables. Not only is it useful when writing a SQL query to periodically check the values of a variable, but it is also helpful when troubleshooting complex SQL scripts that don't seem to be returning the desired values.

13.1.2 Rules for user-defined variables

There are a few rules about variables to note before we go any further. There aren't very many and they aren't difficult to remember, but they are crucial to making sure we use variables correctly.

THE FIRST CHARACTER OF A VARIABLE NAME MUST BE @

The use of @ in the variable name tells the RDBMS you are working with a variable.

THE REMAINING CHARACTERS IN A VARIABLE NAME MUST BE ALPHABETIC OR NUMERIC

As we can see with the use of @, non-alphanumeric characters can have special meanings in SQL. Use only letters and numbers in your variable names.

VARIABLE NAMES CAN BE NO MORE THAN 64 CHARACTERS

We want to be descriptive with our variable names so that others who read our SQL can easily understand their intended purposes, but if the name of any of your variables is anywhere near 64 characters, you are probably being a bit too descriptive.

VARIABLE NAMES ARE NOT CASE-SENSITIVE

If you declare a variable named @Variable, then any usage of @VARIABLE, @variable, or @VaRiAbLe will all refer to the same variable.

A USER-DEFINED VARIABLE CAN HOLD ONLY A SINGLE VALUE

You can't include multiple values, although you can change the value of a variable throughout your SQL if you so desire.

A USER-DEFINED VARIABLE EXISTS ONLY FOR THE DURATION OF THE CONNECTION

Databases and tables *persist*, which means they exist until they are explicitly removed. Unlike those objects, variables do not persist, so once you close your MySQL Workbench or any other tool you use for connecting to a database, any variables you have declared will no longer exist.

13.1.3 Using your first user-defined variable

With all these rules out of the way, we are ready to put variables to use. Let's write some SQL to use a variable to help select the title ID, title name, and publication date from the title table, with the results shown in figure 13.2:

```
SET @TitleName = 'The Sum Also Rises';
SELECT
    TitleID,
    TitleName,
    PublicationDate
FROM title
WHERE TitleName = @TitleName;
```

TitleID	TitleName	PublicationDate
108	The Sum Also Rises	2021-11-12 00:00:00

Figure 13.2 The title ID, title name, and publication date from the title table for “The Sum Also Rises”, as filtered using a user-defined variable.

Now, we have this bit of SQL, which admittedly is a rather short piece of code. Imagine though, if we had much more SQL that needed to be executed for a given title—perhaps finding the number of titles sold or the states with customers who purchased the title. Whatever the case, if we set and filtered on a variable throughout the query as we did earlier, then to query a different title we would need to make the change in only one part of our SQL.

Here's how that looks in practice. Let's change the variable to 'Pride and Predicates' and execute our query again, and see the results shown in figure 13.3:

```
SET @TitleName = 'Pride and Predicates';
SELECT
    TitleID,
    TitleName,
    PublicationDate
FROM title
WHERE TitleName = @TitleName;
```

TitleID	TitleName	PublicationDate
101	Pride and Predicates	2015-04-30 00:00:00

Figure 13.3 The title ID, title name, and publication date from the title table for “Pride and Predicates”, as filtered using a user-defined variable.

Again, this is a simple bit of SQL, but hopefully you can start to see the power of using variables to make your script more flexible and reusable by using variables. Variables are used in nearly every programming language, and we've got plenty of examples in this and subsequent chapters of how we can use them effectively.

TRY IT NOW

Declare a variable with a name of your choosing, and then use it to select the title ID, title name, and publication date from the title table.

13.2 Filtering with variables in FROM and HAVING clauses

Let's look at some practical ways to use variables in your SQL. Suppose we wanted the date of every order of any particular title. We can use a variable to do this, and in this case, we will start with “The Sum Also Rises”. With table joining logic we've used before, we could write something like this, with the results shown in figure 13.4:

```
SET @TitleName = 'The Sum Also Rises';

SELECT
    oh.OrderDate
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE t.TitleName = @TitleName;
```

OrderDate
2021-11-13 00:00:00
2021-11-28 00:00:00
2022-03-10 00:00:00

Figure 13.4 The order date for any order that included the title “The Sum Also Rises”

We can change that title name to any other valid title and return the corresponding set of order dates. As you might imagine, if for some reason our variable is set to a value that is not included in the title table, our result set would be 0 rows.

Another common way to use variables is to find information about orders from a particular day, week, month, or year. Let's find the names of all customers who placed any orders for any titles in November of 2021, as shown in figure 13.5. In this case, we will use two variables to represent the start date and end date of our range:

```
SET @DateStart = '2021-11-01',
    @DateEnd = '2021-11-30';

SELECT
    c.FirstName,
    c.LastName,
    oh.OrderDate
FROM customer c
INNER JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
WHERE oh.OrderDate BETWEEN @DateStart and @DateEnd;
```

FirstName	LastName	OrderDate
Keanu	O'Ward	2021-11-13 00:00:00
Monica	Newgarden	2021-11-28 00:00:00

Figure 13.5 The first name and last name of any customer who placed an order in November of 2021, along with the order date

Interestingly, we can put this filtering in a different part of the predicate. Instead of having this in the WHERE clause, we can make this a condition in the JOIN. This isn't typically how filtering is done in SQL, although you may notice it in other people's code. Here is what the previous query would look like if we filtered on our variables in the FROM clause, as part of a JOIN condition:

```

SET
    @DateStart = '2021-11-01',
    @DateEnd = '2021-11-30';

SELECT
    c.FirstName,
    c.LastName,
    oh.OrderDate
FROM customer c
INNER JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
    AND oh.OrderDate BETWEEN @DateStart and @DateEnd;

```

We can also use a variable to see how many titles have sold above a specific quantity. We can apply the aggregation techniques we learned in Chapter 12 here, with a HAVING clause as a filter that uses a variable. For example, let's get a list of all the title names that have sold 10 or more copies, as shown in figure 13.6:

```

SET @MinimumQuantitySold = 10;

SELECT
    t.TitleName,
    SUM(oi.Quantity) as TotalQuantitySold
FROM orderitem oi
INNER JOIN title t
    ON oi.TitleID = t.TitleID
GROUP BY t.TitleName
HAVING SUM(oi.Quantity) >= @MinimumQuantitySold;

```

TitleName	TotalQuantitySold
Pride and Predicates	25
The Join Luck Club	13
Catcher in the Try	11
The DateTime Machine	13

Figure 13.6 The title name and total quantity sold of all titles that have sold at least 10 copies.

The results show four titles that meet the threshold of at least 10 copies. If we want to change the threshold of our filter to another value, all we need to do is change the value of the @MinimumQuantitySold variable.

13.3 Assign an unknown value to a variable

One of the useful aspects of variables is that we can create and use one even when we don't know the explicit value we want to filter on. For instance, if I asked you what the value for title ID is for "The Sum Also Rises" title, would you know it? Honestly, I wrote everything in this database, and I can't even recall that value from memory.

Although we may not have memorized the title ID values, we've used them countless times in our queries, as these are the values that constitute the relationship between the orderheader and the title tables.

13.3.1 A quick review of how a query works

Let's take a moment to consider how the title ID is used in the first query from section 13.2, and how we can use one variable to collect the value for another variable. Here's that query, which looks for the order dates of the title "The Sum Also Rises":

```
SET @TitleName = 'The Sum Also Rises';

SELECT
    oh.OrderDate
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE t.TitleName = @TitleName;
```

Let's walk through the joins in this table, starting at the bottom and working our way up. Why this way? Because your RDBMS will probably start finding your results by filtering rows in the title table. Filtering typically means fewer rows to read, and fewer rows to read means fewer rows to join with other tables, which will be more efficient than reading all the rows in all the tables, joining them, and then applying the filtering.

In this query, we used the variable @TitleName to find any rows in the title table that matched our query, which happens to be one row. We then join that row to any related rows to orderitem via the matching TitleID values, and then join to any related rows in orderheader using the matching OrderID values in both tables. Once we have related values through all tables, we can now select the values for OrderDate to determine when the titles were ordered.

13.3.2 Assigning an unknown variable with SELECT

Now, this is a relatively simple query with a couple of joins. It's important to note, though, that joins require work by the RDBMS, as it has to read and relate the data in different tables. Fortunately, we can reduce the number of joins in our query in the preceding section by creating a variable for the TitleID values, since we know there is only one value in the title table that matches the name value for "The Sum Also Rises".

We can find the value for TitleID, which is unknown, by declaring our variable a bit differently. We're going to use a SELECT statement instead of SET, since we are selecting a value from an existing table to be used in our variable.

Before we do that, let's take a step back and imagine that we wanted to return the value for TitleID instead of using it to assign a value to our @TitleID variable. We might write SQL like this, using a @TitleName variable for the name of the title:

```
SET @TitleName = 'The Sum Also Rises';

SELECT TitleID
FROM title
WHERE TitleName = @TitleName;
```

TRY IT NOW

We keep talking about this value, so go ahead and execute this query to finally know the TitleID value for "The Sum Also Rises".

We can use the same logic from this query to assign the value of this TitleID to a new @TitleID variable, with the output for the assigned value shown in figure 13.7:

```
SET @TitleName = 'The Sum Also Rises';

SELECT @TitleID := TitleID
FROM title
WHERE TitleName = @TitleName;
```

@TitleID :=
TitleID
108

Figure 13.7 Selecting an unknown value for TitleID into @TitleID will result in the output showing the value that was selected. In this case, the value is 108.

The FROM and WHERE clauses are identical to the previous query, but the SELECT clause looks unlike anything we've done so far. With the logic of `SELECT @TitleId := TitleID` we are able to do two things at once with our SELECT clause: select the TitleID value and assign that value to our `@TitleID` variable.

Also notice that we have used the `:=` operator instead of `=` in our SQL. Recall that earlier we mentioned that you can use `=` or `:=` when assigning a value to a variable using SELECT. When assigning a value to a variable using SELECT in MySQL, you can only use the `:=` operator.

WARNING As noted with the SET keyword previously, this method of assigning an unknown value to a variable will be different for nearly every RDBMS. Although fundamentally similar, it will be important to know the correct syntax used by whatever RDBMS you are working with.

One other interesting side effect of assigning a value to a variable with SELECT is that there is now an output of the results in the Results panel. This is because SELECT statements result in output in MySQL, and in this case the result shows the value that was assigned to the variable, with the expression used in the SELECT statement as the column header.

13.3.3 Performance considerations with variables

All right, now that we've learned how to assign a value to a variable using SELECT, let's see how this looks with the overall query that was intended to find the order dates for "The Sum Also Rises" as shown in figure 13.6:

```

SET @TitleName = 'The Sum Also Rises';

SELECT @TitleID := TitleID
FROM title
WHERE TitleName = @TitleName;

SELECT
    oh.OrderDate
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
WHERE oi.TitleID = @TitleID;

```

OrderDate
2021-11-13 00:00:00
2021-11-28 00:00:00
2022-03-10 00:00:00

Figure 13.8 The order date for any order that includes the title “The Sum Also Rises”. This time we used the **SELECT** keyword instead of the **SET** keyword to get the same results as figure 13.4.

This latest query uses two variables but requires only one join in the final statement instead of two. Although this is still a relatively trivial query in terms of the work required of the RDBMS, in queries against larger sets of data, reducing joins as we have done here can offer significant improvements in performance.

NOTE If you look closely at the Results panel, you will see we actually have two sets of results, with two separate tabs at the bottom of the panel. Those shown in figure 13.8 are the results we wrote our SQL to determine, but as you might have guessed, the results in the other “hidden” tab are the output from the first SELECT where the variable value was assigned.

Performance considerations aside, we can also make our code more readable by using **SELECT** to assign our variables.

13.3.4 Troubleshooting considerations with variables

Consider a request to find the title, quantity, and price of the first order in a particular year, such as 2021. To do this, we first need to find the date of the first order in 2021, and then using that value, we find the information about the order placed on that date. The orders in our database are few enough that there are no more than one per day, so this makes this task simpler.

Let's first determine the date of the first order in 2021, as shown in figure 13.9. Now, we haven't covered this yet, but we could use the SET method to assign an unknown value to our variables instead of SELECT. However, to do this we would have to use a kind of subquery, like this:

```
SET @FirstOrderDate = (
    SELECT MIN(OrderDate)
    FROM orderheader
    WHERE OrderDate BETWEEN '2021-01-01' AND '2021-12-31');

SELECT @FirstOrderDate as FirstOrderDate;
```

FirstOrderDate
2021-01-15 00:00:00

Figure 13.9 The order date of the first order placed in 2021, which is only shown because of the SELECT statement. We will use this value later to determine more information about the order placed on that day.

This query produces the correct result, but we can see the value assigned to the variable only if we explicitly use a separate SELECT statement. However, as we've seen previously, using SELECT instead of SET to assign this value will also show us the value assigned to the variable, as shown in figure 13.10:

```
SELECT @FirstOrderDate := MIN(OrderDate)
FROM orderheader
WHERE OrderDate BETWEEN '2021-01-01' AND '2021-12-31';
```

@FirstOrderDate := MIN(OrderDate)
2021-01-15 00:00:00

Figure 13.10 The order date of the first order placed in 2021, shown without a second SELECT statement. The expression used in the SELECT statement is the column header.

Whichever method you use is a matter of personal preference, and this may largely be determined by whether you want to see in the output if the correct value is being used. As you are writing a query, it may be helpful to use the SELECT method to verify that the values assigned to your variables are correct, to avoid incorrect results. Seeing the values shown in the Results panel may give you more confidence in the effectiveness of your SQL.

For now, let's use the method with SELECT to determine the first order of 2021, to select the title, quantity, and price of the order placed on that day, and see the results in figure 13.11:

```

SELECT @FirstOrderDate := MIN(OrderDate)
FROM orderheader
WHERE OrderDate BETWEEN '2021-01-01' AND '2021-12-31';

SELECT
    t.TitleName,
    oi.Quantity,
    oi.ItemPrice
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderDate = @FirstOrderDate;

```

TitleName	Quantity	ItemPrice
The DateTime Machine	1	7.95

Figure 13.11 The title name, quantity, and price of the items in the first order placed in 2021.

It's good to have options in how you use variables in your SQL, and now you should have a better understanding of the pros and cons of using either SET or SELECT to assign value to your user-defined variables.

13.4 Other notes about variables

Before we wrap up and get to the lab exercises, there are a few more points to consider when using variables.

13.4.1 Assigning a literal value using SELECT

Although we didn't cover it, we can use SELECT to assign a literal value instead of SET. There isn't a correct choice, as this is mostly a matter of personal preference, but we have seen throughout the chapter that SELECT offers more options with its ability to work with FROM, WHERE, and other clauses. This would be the syntax to assign a date value:

```
SELECT @SomeDate := '2021-11-30';
```

13.4.2 Assign a value of NULL to a variable

Often we may find that we want to start with a variable that has a null value, and then see if later in our SQL it gets a value assigned. The variable type doesn't matter here until a value gets assigned, in which case the variable type will be changed to the data type of the value. We can assign a null value to a variable with SET or SELECT using either of the following SQL:

```
SET @NullVariableWithSET = null;

SELECT @NullVariableWithSELECT;
```

13.4.3 We can change the type of data used by a variable

In MySQL, variables can have different values assigned to them throughout their use in your SQL. As just noted, the variable data type can change if you start with null but then later assign a string or an integer or another kind of value. There aren't a lot of use cases for changing the data type of a variable, but if you wanted to, you could even assign different data types to a variable throughout your SQL. Here is an example, where the first assigned value is a number and then a string is later assigned:

```
SET @SomeVariable = 1;

SELECT @SomeVariable as FirstValue;

SET @SomeVariable = 'The Sum Also Rises';

SELECT @SomeVariable as SecondValue;
```

Although it is possible to change a variable type throughout your SQL, doing so falls under the category of "things you can do but shouldn't." In fact, this is only noted in case you make a mistake and accidentally reuse a variable more than once, noting that you won't get an error or warning that you have done so.

Now that we have looked into the options for how we can utilize user-defined variables, let's try some lab exercises.

13.5 Lab

1. In this chapter, it was noted that `:=` must be used when assigning a value using `SELECT`. What happens if `=` is used instead?
2. We also noted that you can only assign a single value to a variable. What happens if you execute the following query? Is a value assigned to the variable? If so, what is the value?

```
SELECT @TitleID := TitleID
FROM title;
```

3. Review the final query used in 13.3.4, and update it using variables for start date and end date.
4. Write a query to find total sales dollars (in terms of quantity x price) for any customer, with customer first name and last name as variables.

13.6 Lab answers

1. If you use = instead of := to assign values to a variable in a SELECT statement, the value of the variable will be null. The MySQL RDBMS uses = to test for equality, as we have seen numerous times when using filters and joins. In the SELECT statement, it will determine that the two values are not equal because the variable has no value. *Remember:* null is the absence of data.
2. The value is 108, although if you execute the query, you will see all the values for TitleID in the results. Because only one value can be assigned to the variable, the final value is assigned. For this reason, be careful to only select a single value when using this method to assign values to a variable.
3. You could use SQL like this to add more flexibility to this query, allowing for easy changes at the top to examine different ranges of data:

```
SET
@DateStart = '2021-01-01',
@DateEnd = '2021-12-31';

SELECT @FirstOrderDate := MIN(OrderDate)
FROM orderheader
AND oh.OrderDate BETWEEN @DateStart and @DateEnd;

SELECT
    t.TitleName,
    oi.Quantity,
    oi.ItemPrice
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderDate = @FirstOrderDate;
```

4. There are a few ways to do this, but here is one way:

```
SET @FirstName = 'Chris';
SET @LastName = 'Dixon';

SELECT @CustomerID := CustomerID
FROM Customer
WHERE FirstName = @FirstName
    AND LastName = @LastName;

SELECT
    @FirstName as FirstName,
    @LastName as LastName,
    SUM(oi.Quantity * oi.ItemPrice) as TotalSalesDollars
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
WHERE oh.CustomerID = @CustomerID;
```

14

Querying with functions

In Chapter 12 we looked at a handful of functions—commands that perform some sort of pre-defined calculation. We looked specifically at some basic aggregate functions that allow us to quickly calculate things like the sum total of a range of values, as well as the minimum, maximum, and average values for a given range.

In this chapter, we are going to examine even more functions that open more possibilities in SQL, including those that allow us to select and filter specific string, date and time, and other informational values. First though, let's take a broader look at when we should and shouldn't use functions.

14.1 The issues with functions

Functions are incredibly useful for selecting specific parts of values, calculating values, and even manipulating values in SQL. They are a bit like magic spells that we can perform by adding only an extra word in our SQL. However, there are two big issues with functions that we need to discuss before we start using them throughout our queries.

14.1.1 Function commands vary for each RDBMS

The core keywords and clauses we have used up until now are, for the most part, universal. When we write SQL using SELECT, FROM, WHERE, and GROUP BY, we know that it's going to work not just in MySQL but also in any RDBMSs we may be using. Functions, however, are not universal, and many of the functions we examine in this chapter will have some sort of variation for one or more RDBMSs.

Now, that doesn't mean the functions you will learn and practice are only for MySQL, as many of these functions will work in another RDBMS. But it does mean that if you try to use them in another RDBMS and encounter a syntax error, you will likely need to do a little research to see what the correct keyword is in that particular RDBMS.

That said, I will do my best to note these variances throughout this chapter, as this can be a potential obstacle if you take your new SQL skills to another RDBMS.

14.1.2 Function commands can be inefficient

Recall that we noted at the end of Chapter 12 how the DISTINCT keyword actually has to do extra work by reading all the values in a range and then returning the requested values without duplicates. We noted how it should be used very carefully with large sets of data because we don't want to use server resources unnecessarily.

Depending on their usage, nearly all functions that we will discuss may also need to read all the values in a range. Although functions are incredibly useful and appear to be shortcuts to achieving a desired output, we need to be mindful of their usage with large sets of data. While using functions with a large set of data can get us the correct answers, they may use more resources and therefore may not be the most efficient way to use SQL to get an answer.

Again, these warnings aren't meant to discourage you from using functions. You just need to be aware of their limitations and effects. Now, let's see how we can use functions to increase our SQL skills.

14.2 String functions

String functions allow us to select parts of string data, which is often required when we need to present data in a way that is different from how it is stored in the database. Examples of these kinds of requirements are returning the names of customers in all uppercase for a mailing list and eliminating unnecessary leading or trailing spaces from a value. Let's look at these right now.

14.2.1 Case functions

The first string function we will try is the UPPER function, which will convert a string of characters to all uppercase characters. We will use it here to view just the customers in California, which are identified by the State value of CA and shown in figure 14.1. We will also include the actual values stored for first name and last name for comparison:

```
SELECT
    FirstName,
    LastName,
    UPPER(FirstName),
    UPPER(LastName)
FROM customer
WHERE State = 'CA';
```

FirstName	LastName	UPPER(FirstName)	UPPER(LastName)
Cora	Daly	CORA	DALY
Tara	Di Silvestro	TARA	DI SILVESTRO
Margaret	Montoya	MARGARET	MONTOYA

Figure 14.1 The first and last names of all customers in California, first as they exist in the customer table and then in all uppercase using the **UPPER** function.

The syntax for usage of **UPPER**, as is the case for most functions, is to call the function and then have the column name, variable, or other value where the function is applied be contained inside of parentheses. For this reason, you will often see functions referred to with parentheses in the function name, such as **UPPER()**.

Although you may have observed that the third and fourth columns are now all uppercase characters as expected, also notice the names of the columns returned. They are the columns as they appear in our **SELECT** clause, which will be the default name if one is not assigned. Let's run the query again, but this time with assigned column names with a prefix of "Upper" with the results shown in figure 14.2:

```
SELECT
    FirstName,
    LastName,
    UPPER(FirstName) as UpperFirstName,
    UPPER(LastName) as UpperLastName
FROM customer
WHERE State = 'CA';
```

FirstName	LastName	UpperFirstName	UpperLastName
Cora	Daly	CORA	DALY
Tara	Di Silvestro	TARA	DI SILVESTRO
Margaret	Montoya	MARGARET	MONTOYA

Figure 14.2 The first and last names of all customers in California, as they exist in the customer table and as all uppercase using the **UPPER** function with defined column names.

That's definitely more readable, and we could use similar logic if we wanted to use another function to make all the characters in the columns lowercase. For that, we can use **LOWER** instead of **UPPER** and see the results in figure 14.3:

```

SELECT
    FirstName,
    LastName,
    LOWER(FirstName) as LowerFirstName,
    LOWER(LastName) as LowerLastName
FROM customer
WHERE State = 'CA';

```

FirstName	LastName	LowerFirstName	LowerLastName
Cora	Daly	cora	daly
Tara	Di Silvestro	tara	di silvestro
Margaret	Montoya	margaret	montoya

Figure 14.3 The first and last names of all customers in California, as they exist in the customer table and as all lowercase using the LOWER function with defined column names.

There aren't a lot of reasons to present values with all lowercase, but if it is required, this function is available to use.

14.2.2 Trim functions

The other example we want to try is removing leading or trailing spaces. We have a few options for this, with three separate functions: RTRIM, LTRIM, and TRIM. The RTRIM function will remove all trailing spaces from a value, also stated as all spaces to the right of the last non-space character. LTRIM will remove all leading spaces of a value, or all spaces to the left of the first non-space character. TRIM will be the same as applying both LTRIM and RTRIM on a value, removing all leading and trailing spaces.

NOTE You could actually use two functions on the same value, such as SELECT (RTRIM(LTRIM(SomeValue))). You just need to make sure your logical order is correct, as the innermost function will be executed first. In fact, you may encounter SQL exactly like this from folks who lacked either the knowledge or ability to use the TRIM function to remove both leading and trailing spaces.

Let's try these functions using a variable with leading and trailing spaces. This may seem nonsensical, but I assure you, if you ever need to program an interface for manual data entry then you will have to deal with leading and trailing spaces in the data.

Our variable will have three leading spaces and two trailing spaces. We will assign column names as well in our query, which shows the trimmed results in figure 14.4:

```
SET @Word = '    word  ';
SELECT
    @Word as WordAsEntered,
    LTRIM(@Word) as WordLTrim,
    RTRIM(@Word) as WordRTrim,
    TRIM(@Word) as WordTrim;
```

WordAsEntered	WordLTrim	WordRTrim	WordTrim
word	word	word	word

Figure 14.4 The results of selecting the character string “ word ”, which has three leading spaces and two trailing spaces, with the three trim-related functions. LTRIM removes the left leading spaces, RTRIM removes the right trailing spaces, and TRIM removes both leading and trailing spaces.

Admittedly, we can't see these results for removing the trailing spaces too well just by looking at them. However, we can use another function to verify that those leading and trailing spaces have been removed: LENGTH. This function will return the length in terms of the number of characters including spaces for each of our values.

We're going to have to do a little math, but it's basic addition and subtraction. The word *word* has four characters, and if we add three leading spaces and two trailing spaces, then the length of our word as entered should be nine characters.

If we trim the three left (leading) spaces, then our LTRIM value should be six (9 minus 6). If we trim the two right (trailing) spaces, then our RTRIM value should be seven (9 minus 2). And if we trim all leading and trailing spaces, we should just have a length of four characters for the word *word*.

Let's test this with SQL and the LENGTH function by wrapping it around each of our functions we have used for trimming spaces, showing the results in figure 14.5. That's right—we can execute a function from inside another function. When doing this, though, remember that the innermost function always gets executed first:

```
SET @Word = '    word  ';
SELECT
    LENGTH(@Word) as WordAsEnteredLength,
    LENGTH(LTRIM(@Word)) as WordLTrimLength,
    LENGTH(RTRIM(@Word)) as WordRTrimLength,
    LENGTH(TRIM(@Word)) as WordTrimLength;
```

WordAsEnteredLength	WordLTrimLength	WordRTrimLength	WordTrimLength
9	6	7	4

Figure 14.5 The results of selecting the length of the strings with the LENGTH function, after applying different trim functions to the string “ word ”. Since the removal of leading and trailing spaces can be difficult to see, we can use LENGTH to validate the results of using the trim functions.

Again, trimming data in this way is important because you generally don't want to store or display data with leading spaces, as those spaces not only take up unnecessary space in your database but also can cause issues with common tasks such as filtering and sorting.

NOTE In SQL Server, there is no LENGTH function. Instead, use LEN to find the length of a string or another expression.

14.2.3 Other string functions

Although each RDBMS has its own set of functions, there are some functions that have such common use cases that they are available in nearly all RDBMSs. Table 14.1 lists a handful of the string functions available in nearly every RDBMS.

Table 14.1 Common string functions and definitions available in most RDBMSs

Function name	Description
LEFT	Gets a specified number of leftmost characters from a string.
REPLACE	Searches and replaces a substring of values in a string.
RIGHT	Gets a specified number of rightmost characters from a string.
SUBSTRING	Gets a substring starting from a specified position with a specific length.

14.3 Date and time functions

We can not only parse string values with functions but also do the same with *date and time values*. There are appropriately named functions for determining the YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND in most RDBMSs, which can be useful when we want to find information based on one or more parts of the date.

14.3.1 Date functions that return numeric values

For example, if we wanted to find a list of all order IDs from 2015 like the one shown in figure 14.6, we could use the YEAR function in a query that checks the year of all orders in orderheader and returns the requested data. Let's include order ID and order date in our query:

```
SELECT
    OrderID,
    OrderDate
FROM orderheader
WHERE YEAR(OrderDate) = 2015;
```

OrderID	OrderDate
1001	2015-06-01 00:00:00
1002	2015-06-15 00:00:00
1003	2015-07-03 00:00:00
1004	2015-08-12 00:00:00
1005	2015-09-05 00:00:00
1006	2015-11-02 00:00:00
1007	2015-11-15 00:00:00
1008	2015-11-22 00:00:00

Figure 14.6 The results of all order id and order dates from orders placed in 2015. We got these results by using the YEAR function.

We could do the same thing if we wanted to work with one of the other six functions relating to date time, and I'm sure you're already thinking about how we can use variables in this kind of search to add even more flexibility for filtering.

Or maybe you are starting to consider that we could also use the functions to select parts of a date and time value. Let's use all of these functions on the order date of the first order to see how this looks in figure 14.7. The first order has an order ID of 1001:

```
SELECT
    OrderDate,
    YEAR(OrderDate),
    MONTH(OrderDate),
    DAY(OrderDate),
    HOUR(OrderDate),
    MINUTE(OrderDate),
    SECOND(OrderDate)
FROM orderheader
WHERE OrderID = 1001;
```

OrderDate	YEAR(OrderDate)	MONTH(OrderDate)	DAY(OrderDate)	HOUR(OrderDate)	MINUTE(OrderDate)	SECOND(OrderDate)
2015-06-01 00:00:00	2015	6	1	0	0	0

Figure 14.7 The results of all the date and time parts from the order date of the first order in the orderheader table.

These functions may not seem helpful now, but I assure you that the next chapter suggests some very practical ways to use these date and time parts to determine the results of various calculations. Also, these aren't the only date and time functions, as there are two others that may interest you: DAYNAME and MONTHNAME.

14.3.2 Date functions that return string values

The DAYNAME and MONTHNAME functions give us a little more information about date values by returning string values for the names of the month and day of a given date. Although month may seem obvious if you know the numeric value, it's highly unlikely you might remember the name of the day of the week on which this order was placed. Let's modify the previous query to use these functions so we can find out, with the results shown in figure 14.8:

```
SELECT
    OrderDate,
    YEAR(OrderDate),
    MONTHNAME(OrderDate),
    DAY(OrderDate),
    DAYNAME(OrderDate)
FROM orderheader
WHERE OrderID = 1001;
```

OrderDate	YEAR(OrderDate)	MONTHNAME(OrderDate)	DAY(OrderDate)	DAYNAME(OrderDate)
2015-06-01 00:00:00	2015	June	1	Monday

Figure 14.8 The results of using the MONTHNAME and DAYNAME to determine the names of the month and day of the first order in the orderheader table.

Remember these functions when you find yourself having to compile reports that need either a date time value formatted a certain way or the name of the month or day specified.

14.3.3 Other date and time functions

Table 14.2 lists some of the more common date and time functions available in nearly every RDBMS.

Table 14.2 Common date and time functions available in most RDBMSs.

Function name	What It Gets
DATE	Only the date from a date and time value.
DAYOFWEEK	The numeric day of the week for a date value.
DAYOFYEAR	The numeric day of the year for a date value.
LAST_DAY	The last date of the month for a date value.
QUARTER	The quarter of the year for a date value.
TIME	Only the time from a date and time value.
WEEKOFYEAR	The numeric week of the year for a date value.

As you can imagine, these functions all have many potential uses for finding out information about date and time values stored in a database. But what if you need to find information about right now, such as the who, where, and when of a query? Fortunately, there are functions to answer these questions as well.

14.4 Informational functions

Each RDBMS will have its own set of ways to answer the who, where, and when of a query, although there are some common functions that are typically used. Let's start with the when, as in When does a query occur?

14.4.1 Date and time information

To determine when right now is, we commonly use the function CURRENT_TIMESTAMP. This function grabs the current time on the server where your database is located, which in this case is most likely the computer you are using for your local installation of the sqlnovel database. The format of the date and time will be similar to what we have seen with the date and time values so far, with a format of [year-month-day hour:minute:second].

Note that when using CURRENT_TIMESTAMP, you still need to use the parentheses like you would with any function, even though you don't pass a value:

```
SELECT CURRENT_TIMESTAMP() AS RightNow;
```

I'm not going to show you the results, since my results when I wrote this would be different than when you run it. And those results will be different every time you execute this.

TRY IT NOW

Determine the current time on your database server using the CURRENT_TIMESTAMP function.

This isn't the only function related to the current time. If you only need the day or the time, most RDBMSs will have the functions CURRENT_DATE and CURRENT_TIME as well. You can try these with a query like this:

```
SELECT
    CURRENT_DATE() AS CurrentDate,
    CURRENT_TIME() as CurrentTime;
```

Since CURRENT_TIMESTAMP is a bit longer than most function names, many RDBMSs will have another function that does the same thing as CURRENT_TIMESTAMP but with fewer letters. For MySQL, this is the NOW function. You can use this SQL to confirm this and see that they both return the same value:

```
SELECT
    CURRENT_TIMESTAMP() AS RightNow,
    NOW() AS AlsoRightNow;
```

One last thing to note about these functions is that they can be used with other functions we have covered. For example, if you need to return the name of today's day of the week, you could always determine it with a query like this:

```
SELECT DAYNAME(NOW()) AS CurrentDayOfWeek;
```

Now, let's move on to a final set of functions that tell us who and where we are.

14.4.2 Connection information

Although we are working with only one database throughout this book, in professional experience you will likely be connecting to multiple databases at various times to query different data. If you are ever uncertain of which database you are connected to, you can always identify it with the DATABASE function and see a result like that in figure 14.9:

```
SELECT DATABASE();
```

DATABASE()
sqlnovel

Figure 14.9 The results of selecting the current database used by the connection, which is the sqlnovel database.

Additionally, you may find yourself using more than one login to connect to a database. This can happen when you change between your own personal login to another used by a specific application or report system, often for testing. We can determine the username being used by our connection with the CURRENT_USER function.

```
SELECT CURRENT_USER();
```

NOTE MySQL has both the USER and CURRENT_USER functions, which both return information about the username. Most RDBMSs will include CURRENT_USER but not USER.

And finally, as we noted in the installation directions referenced at the start of the book, there are always periodical updates to the MySQL database engine, which means when connecting to a database, it is rarely obvious what the version is. If we want to see the version, there is a VERSION function:

```
SELECT VERSION();
```

The version will be returned in a string of three numbers separated by periods, and the first number is the main version. For the exercises in this book, you should be using version 8 or later.

That should be enough new functions for now, but in the next chapter we will discover even more functions that allow us to manipulate values and perform calculations in many practical ways.

14.5 Lab

1. We noted that most functions take a parameter of some kind, but we didn't use any parameters with CURRENT_TIMESTAMP. What happens if you pass a value to CURRENT_TIMESTAMP, such as the number 2?
2. Using the date functions discussed, how can we determine a count of how many orders were placed on a Monday?
3. What two variables can we use to determine the longest title name in the title table? How can we write a query to determine this?

14.6 Lab answers

1. The CURRENT_TIMESTAMP function will accept integers as parameters, which determine the precision used in the value of date and time returned. Adding the 2 as shown below will additionally return milliseconds of the date and time.

```
SELECT CURRENT_TIMESTAMP(2);
```

2. We can use the DAYNAME function, which will allow us to filter rows in the orderheader table that have an order date that happened on a Monday.

```
SELECT COUNT(OrderID) as MondayOrders  
FROM orderheader  
WHERE DAYNAME(OrderDate) = 'Monday';
```

3. We can use the MAX and LENGTH functions to determine the longest title name in the title table.

```
SELECT MAX(LENGTH(titleName))  
FROM title;
```

The next part might be a bit more challenging because it involves a subquery. We can filter on titles with the length determined by the previous query, by using a subquery in the predicate to determine the name of the longest title that has a length matching the MAX length.

```
SELECT TitleName  
FROM title  
WHERE LENGTH(TitleName) =  
      (SELECT MAX(LENGTH(TitleName))  
       FROM title);
```

15

Combining or calculating values with functions

In the previous chapter, we looked at several functions that allow us to return parts of data. In this chapter, we will look at a few more functions that let us combine values in different ways, and even perform calculations. Depending on the nature of the data you are working with, I'm sure you'll find some functions in this chapter that you will be using frequently.

For example, if you work with address data, how can you combine all the columns for street, city, and more into a single column? Or if you work with financial reports, how can you make all your calculations correctly show the desired precision of currency?

We're going to look at these scenarios and more in this chapter. Let's start with how we can use functions to combine values.

15.1 Combining string values

We haven't discussed it yet, but you can use SQL to perform basic calculations, such as addition. Here's an example of basic addition, with the result shown in figure 15.1:

```
SELECT 1 + 1;
```

1
+
1
—
2

Figure 15.1 The results of calculating $1 + 1$ using SQL

This is great news if you are working exclusively with numeric data, but what if you need to combine string values? Unfortunately, as we can see in figure 15.2, using the plus sign doesn't allow us to combine string values to a desired result:

```
SELECT 'I' + ' ' + 'love' + ' ' + 'books!';
```

T + '' + 'love' + '' +
'books!'
0

Figure 15.2 The results of combining string values with the plus operator, which does not combine all the values into a string

Instead of getting a result of "I love books!" we have a result of 0. This result indicates the calculation couldn't be completed because MySQL doesn't know how to mathematically "add" words together.

NOTE If you are using SQL Server, using the plus sign will work with combining strings. However, this won't work for most RDBMSs.

The specific verb describes what we are trying to do here—combining two or more string values to form a single value, is *concatenate*, and it's important to know because the function we will use to concatenate our string values is CONCAT.

15.1.1 CONCAT

If we want to concatenate string values, we can use the CONCAT function to combine multiple values by specifying the list of values separated by a comma. For the previous query, we would use this function in the following way and get the desired result shown in figure 15.3:

```
SELECT CONCAT('I', ' ', 'love', ' ', 'books!');
```

```
CONCAT('I', ' ', 'love', ' ',  
'books!')
```

I love books!

Figure 15.3 The results of using the `CONCAT` function to create a single string value from multiple string values to form the “I love books!” output

This example may seem a bit silly, but as you’ll soon see, this is a *powerful* function.

Of course, string values don’t just exist in the literal values like the ones we used in that query. They exist in the columns of tables or in variables. We can combine any of these string values using `CONCAT` just as easily. For example, let’s create a variable for a title review and concatenate it with all the title names in the title table, with the results shown in figure 15.4:

```
SET @Review = ' is a great book!';  
SELECT CONCAT(TitleName, @Review) as TitleReview  
FROM title;
```

TitleReview
Pride and Predicates is a great book!
The Join Luck Club is a great book!
Catcher in the Try is a great book!
Anne of Fact Tables is a great book!
The DateTime Machine is a great book!
The Great GroupBy is a great book!
The Call of the While is a great book!
The Sum Also Rises is a great book!

Figure 15.4 The results of concatenating the values of the title name column in the title table with a string variable to form a single column of output

We can even use `CONCAT` with numeric or date data types. However, when doing so, we need to be aware that all values will be converted to string data types in order to concatenate values with different data types. Doing this can sometimes cause unexpected results in the sorting of concatenations involving numeric or date values.

To demonstrate, let’s combine the price and title name values in the title table as shown in figure 15.5. We will separate the values with a space for readability. We want to sort the output from lowest values to highest, so we will specify ascending order with `ASC` for emphasis:

```
SELECT CONCAT(Price, ' ', TitleName) as PriceAndTitle  
FROM title  
ORDER BY PriceAndTitle ASC;
```

PriceAndTitle
10.95 The Great GroupBy
12.95 Anne of Fact Tables
7.95 The DateTime Machine
7.95 The Sum Also Rises
8.95 Catcher in the Try
8.95 The Call of the While
9.95 Pride and Predicates
9.95 The Join Luck Club

Figure 15.5 The results of the concatenated values of price and title name with a space to separate them, sorted in ascending order of the concatenated value

What happened here? Numerically the values 10.95 and 12.95 should be at the end of an ascending order, but here they appear at the beginning. This is because these numeric values had to be converted to string values for the concatenation, and they are sorted by the order of the characters. In this case, the first character in these concatenated values is 1, which ordinally comes before the first characters of the other values, which are 7, 8, and 9.

We can get the correct sorting in our output by specifying the specific column we want to sort on, which in this case is the price. Even though the price column by itself isn't in the result set, we can still use it when sorting data like the resulting rows shown in figure 15.6:

```
SELECT CONCAT(Price, ' - ', TitleName) as PriceAndTitle
FROM title
ORDER BY Price;
```

PriceAndTitle
7.95 - The DateTime Machine
7.95 - The Sum Also Rises
8.95 - Catcher in the Try
8.95 - The Call of the While
9.95 - Pride and Predicates
9.95 - The Join Luck Club
10.95 - The Great GroupBy
12.95 - Anne of Fact Tables

Figure 15.6 The results of the concatenated values of price and title name with a space to separate them, sorted in ascending order of price

In case you might have forgotten, we talked about this concept a bit back in chapter 4. To reiterate, just because a column isn't in the output doesn't mean we can't sort by that column in the ORDER BY clause. We can sort by any value or combination of values in the title table that is in the FROM clause, provided we aren't aggregating with a GROUP BY clause. This means our concatenated values can be ordered not only by price or title, but also by title ID or publication date.

TRY IT NOW

Use the preceding query to select the price and title name as a concatenated value, but add a '\$' before the price value to indicate the type of currency. If you are still sorting by price rather than by the concatenated value, then the order should still be in the expected ascending value. You can try sorting by title name or even another column in the title table to change the order of the results.

You may not often need to concatenate values of different data types like we just did, but if you work with customer data, you may need to produce output that concatenates first and last names into a single column. This could be for mailing lists, form emails, name badges, and more.

As you might guess, concatenating first and last names would be very easy to do using CONCAT. Let's concatenate the values for first name and last name in the author table, separated by a space and aliased as "AuthorName", with the results shown in figure 15.7:

```
SELECT CONCAT(FirstName, ' ', LastName) as AuthorName
FROM author;
```

AuthorName
Paul Tripp
Doug Li
Jen Strong
Jorge Guerra
Robert Davidson
Gail Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Figure 15.7 The results of the first and last name of the author table, concatenated into a single value and separated by a space

As useful as the CONCAT function is, if we need to concatenate several values using the same separator, there may be an even better function.

15.1.2 CONCAT_WS

Although it is not supported for every RDBMS, most include an additional CONCAT_WS function to make concatenation with a separator a bit easier. The CONCAT_WS function is similar to CONCAT, with the exception that the first value provided is the separator to be used between all other values.

We could produce the same results shown in figure 15.7 with the following query, which uses the CONCAT_WS function like this:

```
SELECT CONCAT_WS(' ', FirstName, LastName) as AuthorName
FROM author;
```

Using CONCAT_WS doesn't make this particular SQL query any shorter, but if we had to separate more than two columns by spaces or some other separator, the CONCAT_WS function can greatly reduce the number of values you would need—as opposed to using CONCAT, because you need to specify the separator only once.

Since there happens to be a middle name column in the author table, let's try using our CONCAT_WS to add it as well and concatenate each author's full name and see the results in figure 15.8:

```
SELECT CONCAT_WS(' ', FirstName, MiddleName, LastName) as AuthorName
FROM author;
```

AuthorName
Paul K Tripp
Doug Li
Jen Strong
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Figure 15.8 The results of using CONCAT_WS to concatenate the first, middle, and last name of all rows in the author table

The CONCAT_WS function has provided an easy concatenation of all author names, which is remarkable if you consider that some of the middle names have null values. CONCAT_WS will automatically account for those nulls and replace them with empty strings when concatenating values, which is something that CONCAT typically doesn't do.

With CONCAT, the nulls are not converted to an empty string, which can be problematic. As discussed in chapter 7, null values represent the absence of data, so anytime we concatenate another value that isn't null to a null value, the result will always be null.

Let's look at this concept in practice. If we attempt to produce the same results as those shown in figure 15.8 by using CONCAT, we're going to be disappointed in the results. As figure 15.9 shows, any row with a null value for middle name is going to return a result of null:

```
SELECT CONCAT(FirstName, ' ', MiddleName, ' ', LastName) as AuthorName
FROM author;
```

AuthorName
Paul K Tripp
NULL
NULL
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
NULL

Figure 15.9 shows the results of using CONCAT to concatenate the first, middle, and last name of all rows in the author table. The rows with null results are caused by null values in the middle name.

If we must account for possible null values when concatenating with the CONCAT function or any other function that doesn't change null values, the SQL language offers an additional function we can use.

15.1.3 COALESCE

The COALESCE function is supported by every RDBMS and can be used to handle null values in concatenation functions. COALESCE takes a list of values that are input to the function similarly to how we have provided a list of values to CONCAT, and returns the first non-null value from the list.

How this will work in our example shown in figure 15.9 will be to use COALESCE with the value for middle name as the first value in the list and then an empty string for the second value. Since we know the empty string isn't null, we can trust that COALESCE will return non-null values for middle name or the empty string for null values.

We will replace the MiddleName selection in the previous query with our use of the COALESCE function as described, which will ensure there are no null values in the results shown in figure 15.10:

```
SELECT CONCAT(FirstName, ' ', COALESCE(MiddleName, ''), ' ', LastName) as AuthorName
FROM author;
```

AuthorName
Paul K Tripp
Doug Li
Jen Strong
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Figure 15.10 shows the results of using CONCAT to concatenate the first, middle, and last name of all rows in the author table. The COALESCE function has replaced the null values for middle name with empty strings.

Though COALESCE did solve the problems created by the null values for middle name, if you look closely you might see that, for those rows, we now have two spaces between first and last name. There's nothing inherently wrong with that, but in this case, using CONCAT_WS returned better-formatted results.

If you find yourself working with an RDBMS that doesn't offer CONCAT_WS and you need to concatenate values like this and change those double spaces to single spaces, you could still do this. You just need to use another common function that can convert a string of some values to another value.

15.2 Converting values

Now that we have combined multiple values to create a new value, let's look at a few more functions we can use to change values. The first one is REPLACE.

15.2.1 REPLACE

The REPLACE function is one we can use to change the occurrence of any combination of one or more characters within a string to some other combination. This combination is known as a *substring* because it is only a part of the overall string being evaluated.

The REPLACE function takes three values of input. First is the string you are going to search, then the substring you want to replace, and then the substring that will be the replacement.

Here's a simple example. If we wanted to change the way the American word *check* is displayed to the British version, *cheque*, we could replace the letters *ck* in the word *check* with *que* as shown in figure 15.11. We could write the following query using the REPLACE function to do this:

```
SELECT REPLACE('check', 'ck', 'que');
```

REPLACE('check', 'ck', 'que')

cheque

Figure 15.11 shows the results of replacing the *ck* in the string *check* with *que* to convert the word from the American English spelling to British English.

Returning to our problem with changing two spaces to one with our concatenated names in section 15.1.3, we can add a REPLACE function to our query that will replace any occurrence of a double space with a single space, with the results shown in figure 15.12:

```
SELECT REPLACE(  
    CONCAT(FirstName, ' ', COALESCE(MiddleName, ''), ' ', LastName)  
    , ' ', ' ') as AuthorName  
FROM author;
```

AuthorName
Paul K Tripp
Doug Li
Jen Strong
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Figure 15.12 shows the results of the concatenated author names, replacing the null values with an empty string using COALESCE, and the resulting double spaces with a single space using REPLACE.

To solve this particular problem, we ended up using three different functions. As you grow in your experience using SQL, this strategy won't be uncommon, since every function does a very specific task, and you may have to think of ways like this to creatively use more than one function in a query.

Let's look at two more common functions for converting values.

15.2.2 CONVERT and CAST

Two functions are commonly used for converting values from one data type to another: CAST and CONVERT. Both of these functions convert a value from one data type to a different specified data type. We saw in chapter 13 that MySQL has some built-in functionality to convert values automatically to different data types, but not every RDBMS has this functionality. For many, you will need to use one of these two functions to handle any type of data conversion.

And though many RDBMSs will offer both of these functions, some will offer only one or the other. Fortunately, MySQL supports both, so we can practice some queries for each function.

Here is a simple example. The current date values are stored with both date and time. The time portion of this data isn't very helpful in our sqlnovel database, since we haven't captured any data for hours, minutes, or seconds. All we have is zeroes for all the time values of publication date. If we wanted to display only the date part of the publication date in the title table, we would need to use one of these functions.

Let's look first at CONVERT, which requires two values: the value we are changing and then the desired data type we want the data converted to. In our example, we want to convert the data type from DATETIME, as it is stored in the title table, to simply DATE. Let's select both the original publication date and our converted value to show the difference in the results in figure 15.13:

```
SELECT
    PublicationDate,
    CONVERT(PublicationDate, DATE) AS PublicationDateNoTime
FROM title;
```

PublicationDate	PublicationDateNoTime
2015-04-30 00:00:00	2015-04-30
2016-02-06 00:00:00	2016-02-06
2017-04-03 00:00:00	2017-04-03
2018-01-12 00:00:00	2018-01-12
2019-02-04 00:00:00	2019-02-04
2019-12-23 00:00:00	2019-12-23
2020-03-14 00:00:00	2020-03-14
2021-11-12 00:00:00	2021-11-12

Figure 15.13 The results of selecting the publication date from the title table, and the publication date converted to remove the time portion of the value

The data types we can change our values to will vary by each RDBMS, but be aware that converting string values such as names to a numeric or date value will not give a useful result usually, and may even return an error.

TRY IT NOW

Use the SQL in 15.2.2 to convert the publication date of the title table to DATE value, and then try converting the title name to a DATE value as well. In MySQL, this should return a null value for the converted title name values, since the string cannot be converted to a valid date.

The CAST function is quite similar to CONVERT but has a slightly different syntax. Instead of passing two separate values to the function, we pass a kind of phrase that uses the word AS instead of a comma. Here is how we would use CAST for the previous example:

```
SELECT
    PublicationDate,
    CAST(PublicationDate AS DATE) AS PublicationDateNoTime
FROM title;
```

Executing this query will return the same results as those shown in figure 15.13.

So why would you use one over the other? Aside from personal preferences, CAST is likely to be preferred because it is considered to be a standard function of SQL, whereas CONVERT is not. Because of this, you are much more likely to have CAST included in the list of functions for a given RDBMS, which means SQL written for one RDBMS that includes CAST is more likely to be usable in another RDBMS.

NOTE Although these two functions effectively do the same thing, the CONVERT function will often have additional functionality that allows for a third value to be passed for formatting the output. Check with the documentation of your RDBMS to see whether CONVERT is supported and has additional options.

15.3 Numeric calculations with functions

In chapter 12 we discovered several aggregate functions, such as MIN, MAX, AVG, and SUM. These aggregate functions are all types of numeric functions, which can be applied to numeric values for various calculations.

Many other numeric functions are available, but most of these are performing specific mathematical calculations, such as the square root of a value, or the logarithm, or the tangent. For now, let's focus on just one mathematical function that can have some practical use cases for most users.

The ROUND function provides an easy way to solve a few common issues where a value needs to be rounded with fewer decimal places—for example, if we need to convert decimal values for currency to integer values for simplicity. Most businesses wouldn't publicly proclaim they sold, for example, \$1,000,000.32 in merchandise. They would instead round the number to \$1,000,000.

Although there hasn't been a million dollars in sales in the sqlnovel database, we can still use ROUND to produce the total sales in whole dollars for a given year without cents, or fractions of a dollar.

Let's see how we would use the ROUND function to calculate the integer value for total sales value for all orders. You might recall we had a calculation of this very value in chapter 12 using the SUM function. Here, we will just add a second column that wraps around that calculation using the ROUND function. Let's select both the sum and the rounded sum to show the difference, with the results shown in figure 15.14:

```
SELECT
    SUM(Quantity * ItemPrice) AS TotalOrderValue,
    ROUND(SUM(Quantity * ItemPrice)) AS TotalOrderValueRounded
FROM orderitem;
```

TotalOrderValue	TotalOrderValueRounded
573.50	574

Figure 15.14 shows the results of the total value of all sales in dollars and cents, along with the same value rounded to an integer.

Note that the rounding in this case made the value increase, since anything equal to or greater than .50 will be rounded to a higher value, whereas any value less than .50 will be rounded to a lower value. We can easily prove this with a simple SELECT statement that will round the value 573.49 to show it will be rounded to a lower value.

TRY IT NOW

Execute `SELECT ROUND(573.49);` to verify this value will be rounded down to 573.

One other thing to note about the ROUND function is that it actually has two parameters. The first is for the number value, as we have used in this chapter. The second parameter, however, is optional and is used to specify the number of places beyond the decimal the number should be rounded. If a value for the second parameter is not passed, it will convert to an integer value (0 spaces beyond the decimal), as we have seen with our examples.

If you work with calculations involving currency, you very likely will need to use ROUND with both parameters. Suppose we need to calculate the added sales tax for a purchase of a title that is \$9.95. If the tax were 5%, we could calculate that amount by multiplying \$9.95 by .05. However, if we do that, the resulting value for the tax value will be more than two decimal places. Since customers cannot be charged fractions of cents, we can pass a value of 2 to the second parameter of ROUND to make the tax value match the currency.

Let's use this query to validate these results, showing both the calculated tax and the rounded tax in figure 15.15:

```
SELECT
    9.95 * .05 AS CalculatedTax,
    ROUND(9.95 * .05, 2) AS CalculatedTaxRounded
```

CalculatedTax	CalculatedTaxRounded
0.4975	0.50

Figure 15.15 shows the results of using ROUND to round 573.49 from two decimal places to one, which rounds the value slightly higher.

If you are required to work with more complicated calculations, your RDBMS likely has dozens of additional functions for mathematical equations.

Table 15.1 lists some of the more common mathematical functions available in nearly every RDBMS.

Table 15.1 Common mathematical functions available in most RDBMSs

Function name	What it produces
ABS	The absolute value of a number
CEIL	The smallest integer not less than a number (rounding up)
FLOOR	The largest integer not more than a number (rounding down)
MOD	The remainder (modulo) of a number divided by another
SQRT	The square root of a number

NOTE In SQL Server, the CEIL function is replaced by CEILING.

So far, we have spent every chapter looking at ways to use SQL to select data from tables in a database and present the resulting output in different ways. In the next chapter, we will look at how to make changes to the actual data in the tables through data manipulation.

15.4 Lab

1. Using the author table, select a single column aliased as “AuthorName” of all author first and last names but with the format of last name, first name—for example, “Iannucci, Jeff”.
2. Write a query where the output is a sentence that declares the publication date of the first title. It can be something like, “The first title was published on 2001-01-30,” except that you use the actual publication date formatted with the date and not the time.
3. It’s a common practice to ignore articles like the word *The* when sorting titles alphabetically. Write a query that will return the title names of all titles in the title table sorted alphabetically in this manner.

15.5 Lab answers

1. We can use the CONCAT_WS function to format the names, with a query like this:

```
SELECT CONCAT_WS(', ', LastName, FirstName) as AuthorName
FROM author;
```

2. Depending on which function you prefer to use, you can accomplish this output in one of several ways. If you used CAST, your query could look like this:

```
SELECT CONCAT('The first title was published on ', CAST(PublicationDate AS DATE), '.')
AS FirstPublicationDate
FROM title
WHERE TitleID = 101;
```

You could accomplish the same output using CONVERT:

```
SELECT CONCAT('The first title was published on ', CONVERT(PublicationDate, DATE), '.')
AS FirstPublicationDate
FROM title
WHERE TitleID = 101;
```

You could also use the DATE function discussed in the last chapter:

```
SELECT CONCAT('The first title was published on ', DATE(PublicationDate), '.') AS  
FirstPublicationDate  
FROM title  
WHERE TitleID = 101;
```

3. This one may be a bit tricky since it requires using the REPLACE function in the ORDER BY clause, which is something we haven't done yet. By doing this, we can replace the word *The* in the title name with an empty string for sorting. Be sure to also include the space after the word *The* to achieve the correct sort order:

```
SELECT TitleName  
FROM title  
ORDER BY REPLACE(TitleName, 'The ', ''');
```

16

Inserting data

For 15 chapters we have examined various ways to read data using the SELECT statement. But all that data we have read using the SELECT statement had to get into the tables somehow, so in this chapter we're going to learn how to insert data.

Throughout this book, we have seen how the syntax of SQL is often like the English language, and when it comes to inserting rows of data, this still holds true, as the new keyword we will be using is INSERT. Let's look at some different ways we can use INSERT to add data to the tables in our database.

16.1 Inserting specific values

The first way we can insert data is by using specific values. The main idea to keep in mind is that when we insert data, we are inserting an entirely new row into an existing table.

Recall back to chapter 2 where we talked about rows, columns, and values. All tables in our RDBMS have rows of data, and each row has a specific set of properties defined by the columns of the table. Each of those properties in the columns is represented by some value of a particular data type, sometimes using NULL to represent the absence of a value for a particular column.

I hope by now the previous paragraph makes total sense to you, especially given all the SQL queries we have written so far. However, if anything about it seems unclear, I encourage you to go back and review chapter 2 to solidify your understanding of rows, columns, and values.

If this all makes sense, we can proceed and insert some data.

16.1.1 Inserting a new row

It has already been noted that we will use `INSERT` as the keyword to insert data, and if we want to insert some specific values, then we will also use the keyword `VALUES`. For our first query we're going to insert a new row in the title table, but before we do that, let's take a look at the data in that table.

In chapter 3, I gave you a warning about using `SELECT *`, but since this table is small, we can use it to save us from typing all the column names in this ad hoc query. We will use this logic a few times in this chapter to quickly glance at the rows of data in different tables:

```
SELECT *
FROM title;
```

TitleID	TitleName	Price	Advance	Royalty	PublicationDate
101	Pride and Predicates	9.95	5000.00	15.00	2015-04-30 00:00:00
102	The Join Luck Club	9.95	6000.00	12.00	2016-02-06 00:00:00
103	Catcher in the Try	8.95	5000.00	10.00	2017-04-03 00:00:00
104	Anne of Fact Tables	12.95	10000.00	15.00	2018-01-12 00:00:00
105	The DateTime Machine	7.95	5500.00	15.00	2019-02-04 00:00:00
106	The Great GroupBy	10.95	0.00	20.00	2019-12-23 00:00:00
107	The Call of the While	8.95	2500.00	15.00	2020-03-14 00:00:00
108	The Sum Also Rises	7.95	5000.00	12.00	2021-11-12 00:00:00

Figure 16.1 The results of all rows and columns in the title table

If we're going to insert a new row into this table shown in figure 16.1, we will need to have values for all columns of data, and we need to use the correct data types. Additionally, for readability, we will specify the columns in the same order as they appear in the table, with `TitleID` first, `TitleName` second, and so on.

Here is the query we will use to insert a new row for the "David Emptyfield" title. We will refer to this kind of query as an `INSERT` statement:

```

INSERT INTO title (
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
)
VALUES (
    109,
    'David Emptyfield',
    9.95,
    0.00,
    10.00,
    '2022-01-16'
);

```

In the preceding query, we are adding the next sequential TitleID (109), the TitleName, and values for Price (9.95), Advance (0.00), Royalty (10.00), and PublicationDate (2022-01-16). If you execute this query, you should see a successful message in the Output panel with a message that reads “1 row(s) affected.”

TRY IT NOW

Write and execute the preceding query. Typing out all the column names of the title table may seem like a pain, but remember you can use the shift key to select all the column names in the navigator window and then drag them into your Query window. If you don’t recall how to do this, you can refer to chapter 3 to refresh your memory.

Something else to notice about our query is how both the column names and the value names are surrounded by parentheses. When we use parentheses, we define the order of both the columns and values used by our INSERT statement.

Interestingly, we could just as easily have written the preceding query with a different column order. The only consideration would be that we need to have the values rearranged in the same order the columns are specified in the INSERT.

Here is an example with the order of our columns reversed. If you have already executed the previous INSERT statement, then don’t execute this query:

```
INSERT INTO title (
    PublicationDate,
    Royalty,
    Advance,
    Price,
    TitleName,
    TitleID
)
VALUES (
    '2022-01-16',
    10.00,
    0.00,
    9.95,
    'David Emptyfield',
    109
);
```

Why would we change the column order for an INSERT statement? Well, we normally wouldn't, as it could look confusing to anyone who would read our SQL statement. However, if you ever find yourself inserting data into a table that has dozens or even hundreds of columns, your SQL might be more readable if you organize column names alphabetically in your INSERT statement instead of the order of the data in the table. Outside of that example, you are probably better off ordering the column names in your INSERT statement in the same order as the table.

16.1.2 Inserting multiple new rows

Another interesting thing about enclosing our values in parentheses is that they indicate all the values used for inserting a single row, which means we also can insert multiple rows if needed. Just as we use a comma to indicate the separate columns and values in our insert statement, we can use a comma to indicate separate sets of values to be inserted as rows.

Here is an example of inserting two more rows into the title table using multiple values in the VALUES portion of our INSERT statement:

```

INSERT INTO title (
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
)
VALUES (
    110,
    'Red Badge of Cursors',
    7.95,
    0.00,
    15.00,
    '2022-03-29'
),
(
    111,
    'Of Mice and Metadata',
    8.95,
    0.00,
    12.00,
    '2022-05-17'
);

```

This example notes only two additional rows, but there's no defined limit to how many rows you could insert into a table. Realistically the only limitation is the amount of storage space used by your database, which is difficult to determine with any kind of general SQL statement and thus out of the scope of the material for this book. That being said, unless you are inserting thousands or millions of rows, you probably don't have to worry about storage space.

NOTE Speaking of general SQL statements, as you review SQL in another RDBMS, you might notice in someone else's SQL the omission of the word `INTO` in `INSERT` statements. That is, the SQL will read "`INSERT [table name] VALUES (...)`" Omitting the word `INTO` is optional in some but not every RDBMS; therefore, it is recommended you practice the habit of including it in your SQL so that you don't have to worry about having syntax errors if you work with multiple RDBMSs.

So far, all our `INSERT` statements in this chapter have involved inserting an entire row or rows of data. As you work with SQL, you will find situations when not only do you not need to insert values for every column, but you may also be programmatically prevented from doing so. Let's look at how we can handle inserting a partial row.

16.1.3 Inserting a partial row

The term *partial row* may sound confusing because we noted in chapter 2 that all rows in a given table must include values for all columns. This is still true; however, there are two considerations that allow us to insert a partial row of data.

The first consideration is if our table includes columns that allow for NULL values. Let's look at the author table for an example of this:

```
SELECT *
FROM author;
```

As we can see in figure 16.2, the MiddleName column in the author table allows for null values, since not all authors will have a middle name. While this nullable column presented challenges when we worked with functions and concatenation in chapter 15, it affords us the option to ignore it when inserting rows.

AuthorID	FirstName	MiddleName	LastName	PaymentMethod
1	Paul	K	Tripp	Cash
2	Doug	NULL	Li	Check
3	Jen	NULL	Strong	Check
4	Jorge	Armando	Guerra	Check
5	Robert	Grant	Davidson	Check
6	Gail	Anne	Shawn	Check
7	Rebecca	NULL	Miller	Check
8	Andy	NULL	Melkin	Direct Deposit
9	Buck	NULL	Fernandez	Cash
10	Chris	NULL	Walenski	Direct Deposit
11	Deepthi	NULL	Mahadevan	Direct Deposit

Figure 16.2 The results of all rows and columns in the author table

If we want to insert a row with a value of NULL for MiddleName, we can ignore the column in our INSERT statement altogether. Doing this will result in entering a value of NULL by default. Here is an example of that kind of insert:

```
INSERT INTO author (
    AuthorID,
    FirstName,
    LastName,
    PaymentMethod
)
VALUES (
    12,
    'Whitney',
    'Miller',
    'Cash'
);
```

We can execute this query without any kind of error, since the MiddleName column allows for values of NULL and will in fact enter that as a default for our new row. If we execute the preceding query and then select all the rows in the author table, we can verify this in the results shown in figure 16.3.

AuthorID	FirstName	MiddleName	LastName	PaymentMethod
1	Paul	K	Tripp	Cash
2	Doug	NULL	Li	Check
3	Jen	NULL	Strong	Check
4	Jorge	Armando	Guerra	Check
5	Robert	Grant	Davidson	Check
6	Gail	Anne	Shawn	Check
7	Rebecca	NULL	Miller	Check
8	Andy	NULL	Melkin	Direct Deposit
9	Buck	NULL	Fernandez	Cash
10	Chris	NULL	Walenski	Direct Deposit
11	Deepthi	NULL	Mahadevan	Direct Deposit
12	Whitney	NULL	Miller	Cash

Figure 16.3 The results of all rows and columns in the author table, including the new row where AuthorID is 12, which has a value of NULL for MiddleName

Not entering a value in this query may seem like we're just being lazy, but you will definitely encounter tables where you want to insert rows that have null values for certain columns. One common example would be if the table has columns for modification date and modification user to track when data was changed and who changed it. As we insert new data, we want those columns to be null to show that the data has not been modified since it was initially added to the table. We will discuss modifying data in greater detail in the next chapter.

Getting back to entering partial rows, the second reason we might need to enter a partial row of data would be if there are default constraints on one or more columns on a table. A *default constraint* is something that the architect of a table has created to set a defined value for a column when data is inserted, which means data for one or more columns will automatically be populated instead of entered by our INSERT statement.

Let's look at our author table again. Notice that the first column is AuthorID, and it features an incremental sequence of numbers from 1 to 12. We need these numbers to be unique for each row because they represent the key values used to relate to data in other tables. We talked about key values in chapter 8.

Because these values need to be unique, database architects will often put a default constraint on the first column, which automatically populates that column with an identity value, which ensures that you and I don't accidentally enter the same identity value for different rows. If we were to do that, the relationships with data in any table using AuthorID would now be ambiguous, as they would have relationships with one or more author rows.

Now, our author table currently has no kind of default constraint for automatically populating the AuthorID column with a default value. We know this because we just manually entered a value of 12 for the row we inserted. If there was a default constraint for an identity value in place, we wouldn't be allowed to enter a value for AuthorID. And if that were the case, as it often is in most database tables with ID values, we would write our INSERT statement to omit the AuthorID column, like this:

```
INSERT INTO author (
    FirstName,
    MiddleName,
    LastName,
    PaymentMethod
)
VALUES (
    'Whitney',
    NULL,
    'Miller',
    'Cash'
);
```

Again, our database isn't set up with a default constraint on the author table, so don't execute this query. Just know you will need to account for these kinds of columns with partial inserts in many real-world databases.

16.1.4 A word of caution about omitting columns

There is another kind of omission in the INSERT statement that you could likely see when you work with someone else's real-world SQL. It is the omission of the table columns in the INSERT INTO portion of the statement.

As an example, you can write an INSERT statement like this and execute it successfully:

```
INSERT INTO author
VALUES (
    12,
    'Whitney',
    NULL,
    'Miller',
    'Cash'
);
```

This may seem tempting, but if you consider what we've discussed in this chapter, then you can already see why this is a dangerous option.

First, this only works if you have values for all the columns in the table and you have the values all in the correct order. If we have one too many or one too few values, the query will fail with an error. And if we have the values in the wrong order, then at best we have created a row with inaccurate data, and at worst we will encounter an error. (Actually, an execution error might be preferable to inaccurate data, as we would still have data integrity.)

Second, this kind of query will start failing if we ever have any change to the underlying table. It's not uncommon for columns to be added to a table, and if we add a new column to our author table, then this query will no longer execute successfully.

Moreover, as we just discussed a few paragraphs ago, this table could have default constraints that prevent us from entering values for certain columns. If our table has any such default constraints, then an INSERT statement without columns specified would fail to execute.

For all these reasons, you should always write INSERT statements that specify column names, and if you see any SQL that doesn't, then see if you can rewrite it to include column names.

With that out of the way, let's move on to more options with inserting data. So far, we have used the VALUES keyword to insert rows of data, but there are a few other ways. The good news is you've already learned how to do them both.

16.2 Inserting a row with a query

When we use VALUES as we have in our INSERT statements thus far, we are saying to the RDBMS, "hey RDBMS, I have these values, and I want to insert them into this table." In both SQL and in English, there are two separate parts to our statement: the location of the insert and the declaration of the values to be inserted.

To use a query instead of VALUES for our INSERT statement, we simply replace the latter part with a SQL query that has columns selected in both order and data type that matches the columns of the table where we are inserting, as defined in the first part of our query.

Since we have already inserted a new title ("David Emptyfield") and a new author ("Whitney Miller"), we can relate this title to this author using the titleauthor table. We haven't talked about this table yet, so let's take a look at it with an ad hoc SELECT * query, with the results shown in figure 16.4:

```
SELECT *
FROM titleauthor;
```

TitleID	AuthorID	AuthorOrder
101	2	1
102	3	1
103	4	1
104	5	1
105	6	1
106	7	1
107	11	1
107	1	2
102	8	1
102	9	2
102	10	3

Figure 16.4 The results of all rows and columns in the titleauthor table

This table has three columns: TitleID, AuthorID, and AuthorOrder. As you might guess, TitleID relates to the TitleID column in the title table, and AuthorID relates to the AuthorID column in the author table. One thing you might notice is that we do not necessarily have unique values for either column in this table. This is because this table represents a many-to-many relationship, as any title could have more than one author, and any author could have contributed to more than one title.

The third column is AuthorOrder, and it refers to the order of the authors as displayed on the cover of a title. This will always be 1 if there is one author, which is what matters for the new row of data we are going to enter.

All right, let's start with a SELECT statement that will allow us to query the tables required. Since this data isn't related yet, we can use subqueries as discussed in chapter 11:

```
SELECT
    (SELECT TitleID FROM title WHERE TitleName = 'David Emptyfield') AS TitleID,
    (SELECT AuthorID FROM author WHERE FirstName = 'Whitney' AND LastName = 'Miller') AS
AuthorID;
```

This query serves as the starting point for our insert. Note that although column aliases are not required, they have been added for readability and to help us match the resulting values to the columns in the titleauthor table.

Let's add the necessary parts to make this query an INSERT statement, including adding a value of 1 for the AuthorOrder column. Here is what our final query will look like:

```

INSERT INTO titleauthor (
    TitleID,
    AuthorID,
    AuthorOrder
)
SELECT
    (SELECT TitleID FROM title WHERE TitleName = 'David Emptyfield') AS TitleID,
    (SELECT AuthorID FROM author WHERE FirstName = 'Whitney' AND LastName = 'Miller') AS
AuthorID,
    1 AS AuthorOrder;

```

Again, the aliasing of the columns in the SELECT clause is for readability, but the important part is that we have the columns in both parts of the query in a matching order.

TRY IT NOW

If you haven't already done so, execute the SQL that inserts "David Emptyfield" to the title table from 16.1.1, "Whitney Miller" to the author table from 16.1.2, and then the preceding INSERT statement that relates them in the titleauthor table. We will use this data in the next chapter.

Now, let's look at another way to insert data, using variables.

16.3 Inserting a row with variables

In chapter 13 we worked with many SELECT queries that utilized variables, which are highly desirable for any kind of repeatable SQL we need to use. We can declare and then use variables in an INSERT statement that uses SELECT to make easily repeatable SQL that only needs variables changed.

Here is an example that we can use to add "A Table of Two Cities" to the title table:

```

SET
    @TitleID = 12,
    @TitleName = 'A Table of Two Cities',
    @Price = 9.95,
    @Advance = 0.00,
    @Royalty = 15.00,
    @PublicationDate = '2022-08-07';

INSERT INTO title (
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
)
VALUES (
    @TitleID,
    @TitleName,
    @Price,
    @Advance,
    @Royalty,
    @PublicationDate
);

```

This is just one way to use variables to insert data, and if you have read chapter 13 then I'm sure you are already thinking about other ways you could use variables to add data to tables in a database.

By learning how to add data by writing INSERT statements, you have now started executing SQL statements that are categorized as data manipulation. *Data manipulation* refers to the process of changing data, and it applies not just to adding data but also to modifying or even removing data. In the next chapter, we will expand our data manipulation skills by learning how we can modify and remove data in our tables.

16.4 Lab

1. Will this SQL, with different alias column names from the table column names, execute successfully?

```

INSERT INTO titleauthor (
    TitleID,
    AuthorID,
    AuthorOrder
)
SELECT
    (SELECT TitleID FROM title WHERE TitleName = 'David Emptyfield') AS TID,
    (SELECT AuthorID FROM author WHERE FirstName = 'Whitney' AND LastName = 'Miller') AS
AID,
    1 AS AO;

```

2. Will this SQL, with no table column names specified, execute successfully?

```

INSERT INTO author
VALUES (
    13,
    'Jeff',
    'Iannucci'
);

```

3. Insert a new row into the promotions table using the following values.
- o PromotionID: 13
 - o PromotionCode: 2OFF2022
 - o PromotionStartDate: May 1st, 2022
 - o PromotionEndDate: May 15th, 2022
4. Insert a new row into the customer table for Gianluca Rossi. The CustomerID should be 21, but all other information will be the same as that of Mia Rossi, whose CustomerID is 20. For this exercise, you should use SELECT instead of VALUES for the insert.

16.5 Lab answers

1. Yes. The column names, whether they are noted in the aliases in the SELECT clause or not, do not need to match the column names in the table where the data is being inserted. What matters is that we have the same number of columns with matching data types.
2. No. Execution of this query will error that the column count doesn't match, since the author table has four columns and this query is attempting to insert values for three values. This is one reason you should always specify the columns in the table where the data is being inserted.
3. Your code should look something like this.

```
INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
)
VALUES (
    13,
    '2OFF2022',
    '2022-05-01 00:00:00',
    '2022-05-15 00:00:00'
);
```

4. Although you could specify all the values being inserted for the new row with literal values, you could also copy the existing values where the CustomerID = 20 for all values except CustomerID and FirstName, like this:

```
INSERT INTO customer (
    CustomerID,
    FirstName,
    LastName,
    Address,
    City,
    State,
    Zip,
    Country
)
SELECT
    21,
    'Gianluca',
    LastName,
    Address,
    City,
    State,
    Zip,
    Country
FROM customer
WHERE CustomerID = 20;
```

17

Updating and deleting data

In chapter 16 we discussed inserting new rows of data into tables, which contained our first exercises of doing something with SQL other than reading data. It was also briefly mentioned that the `INSERT` keyword is one of several used for data manipulation.

In this chapter we will examine two other ways of data manipulation: *updating* and *deleting*. Because SQL is designed to be intuitive for English speakers, we will be working with the keywords `UPDATE` and `DELETE` respectively. Let's start with `UPDATE`.

17.1 Updating values

Updating data is a bit different from inserting data, in that we are manipulating data at the column level instead of the row level. Recall that tables have rows, and rows have properties represented by columns. When we update data in SQL, we are updating the values of those properties, so we are making changes at the column level.

These changes may or may not include all columns in a given row, and they may or may not include updating the values for one or more columns of every row in the table. The point is that we have a lot of options for updating data in SQL, but if this seems confusing, let's look at some examples to help understand these options.

17.1.1 Working with data manipulation in real-time

Before we begin, I want to bring attention to something about SQL and relational databases that often catches people off guard. You may be accustomed to working with a word processor or spreadsheet or another application that contains some kind of data, which saves your changes only when you click a particular button or type a certain series of keys. In these applications, if you make a mistake, you can always go back and undo the changes before you save them.

Unlike those applications, any data manipulation you make in a relational database happens in real-time and is permanent. If you have an “oops” moment, there is no way to undo the changes. They are instantly committed. This may seem alarming, but this is absolutely necessary for an RDBMS to provide up-to-date data quickly and accurately to hundreds or thousands of users who are connected and constantly querying a database.

WARNING Just for extra emphasis, every RDBMS data change we will make in this chapter will happen in real time. There is no save button or keystroke in SQL.

Given that these changes happen instantly and that humans have a knack for making mistakes, the MySQL Workbench has a built-in safety feature called Safe Updates. Safe Updates is enabled by default, and reduces the chances of an accidental data change that can affect every row in a table.

One specific way this is done is that with Safe Updates enabled we can't make any updates that don't specify the key value of a table. Since our tables don't currently have key values, we need to disable this feature in order to make updates. If we don't disable it, our SQL statements will result in errors.

There are a few ways to disable this, but the most complete way is to go to the upper left of the Workbench and select Edit > Preferences. This will open the Workbench Preferences window as shown in Figure 17.1. From here, select SQL Editor from the window on the left, and then scroll down and deselect the checkbox for Safe Updates. After deselecting the box, click OK to close the window and then close and restart the Workbench.

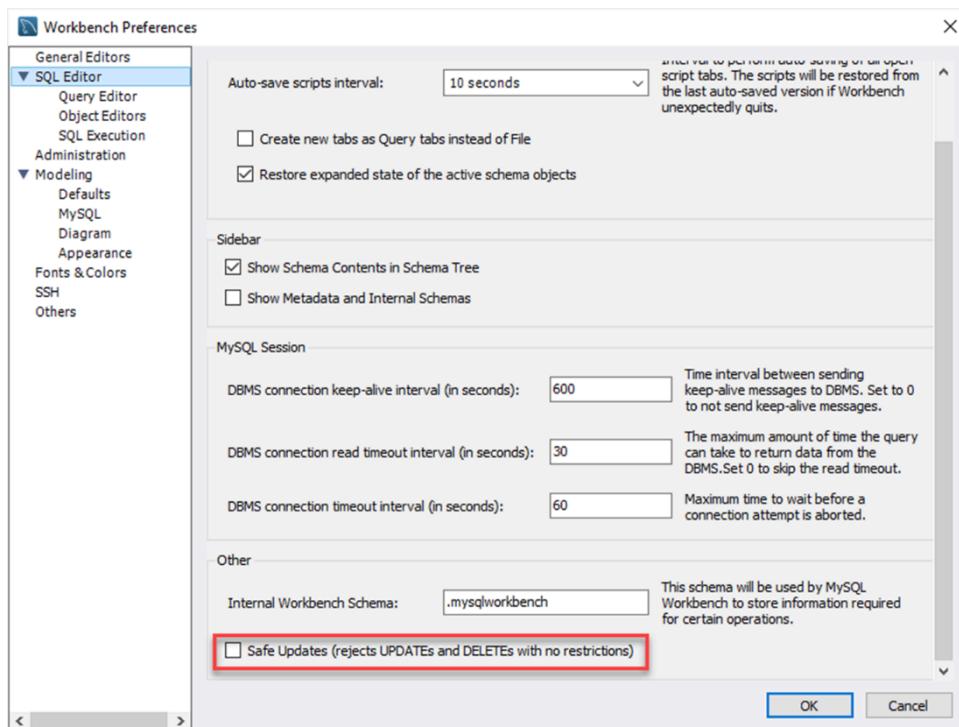


Figure 17.1 The Safe Updates, disabled in the Workbench Preferences

Now that we have disabled Safe Updates, we can carefully begin to make updates to our data.

17.1.2 Requirements for updates

In our first example, we want to update the price of a single title. We want to update the price of "Pride and Predicates" from \$9.95 to \$8.95. Like so many statements in SQL, let's start with an English verbal declaration.

"I would like to update the price of Pride and Predicates to \$8.95."

This is a good starting point verbally, and it's close to what our SQL will be, but we do have to make a few changes to account for table and column names.

"I would like to update the price to \$9.95 where the title name is Pride and Predicates."

Our verbal statement is closer to what our SQL will be like. We have our filter of "where the title name is..." but we need to mention the table where the data is contained. We do this by saying we want to update the table, and then setting one or more column values to some other value.

"I would like to update the title table. I would like to set the price to \$9.95 where the title name is Pride and Predicates."

This is perfect. Let's convert this verbal statement into a SQL statement:

```
UPDATE title
SET Price = 8.95
WHERE TitleName = 'Pride and Predicates';
```

We have one new keyword here, UPDATE, and one we have used before with variables, SET. We use UPDATE to indicate the table where data is to be updated, and we use SET to assign new values in much the same way as we did with assigning values to variables. In fact, this update statement highlights the three requirements for any update, and the order they must be in to be a valid SQL statement.

1. The table that is to be updated, using UPDATE.
2. The column name(s) and new values to be assigned, using SET.
3. The filter condition to determine which rows are to be updated, using WHERE in this case.

WARNING Although technically not a requirement, the WHERE clause is perhaps the most critical of these three. If you do not write *and execute* the filtering condition, then you will update *every* row in the table. Because these changes are in real-time, always take the utmost care to write *and execute* the filter to change only the values in the intended rows.

17.1.3 Updating values in one or more columns

As you may have noticed in the second requirement for update statements, we have the ability to update values in one or more columns with our statement. Much like we use commas to indicate multiple columns in a SELECT statement, we can use commas to indicate two or more desired updates in our UPDATE statement. Just note that all values to be changed in an UPDATE statement must be intended for the same table and meet the same requirements of the filtering condition.

Here is an example where we will update both the Advance and Royalty values to 0 and 10 respectively for the title Pride and Predicates:

```
UPDATE title
SET Advance = 0.00,
    Royalty = 10.00
WHERE TitleName = 'Pride and Predicates';
```

Although we are using the TitleName in our filtering condition, it's more common to use a key value or some other unique identifier for UPDATE statements to ensure we are updating the precise row or rows intended. Given this, let's change our update statement to use the identifier of TitleID, which for Pride and Predicates is 101:

```
UPDATE title
SET Advance = 0.00,
    Royalty = 10.00
WHERE TitleID = 101;
```

NOTE If you have executed the last two statements, you'll notice they both succeeded but with different messages in the Output panel. While the former statement will say "1 row(s) affected" and "Changed: 1", the latter statement will say "0 row(s) affected" and "Changed: 0". This is because there was nothing to change, as the Advance and Royalty values were both already set to the intended values in the UPDATE statement.

We can also use variables to create repeatable code like we did in chapter 16 with INSERT statements. Here is an example where we set all the values back to what they originally were for Pride and Predicates, using the TitleID as the filter condition:

```
SET
@TitleID = 101,
@Price = 9.95,
@Advance = 5000.00,
@Royalty = 15.00;

UPDATE title
SET Price = @Price,
    Advance = @Advance,
    Royalty = @Royalty
WHERE TitleID = @TitleID;
```

TRY IT NOW

If you haven't already, go ahead and execute the UPDATE statements so far in this chapter. Use a query such as `SELECT * from title WHERE TitleID = 101;` in between each of the statements to verify the changes have occurred.

There is one other change we can make with an update, which is the removal of a value. Recall that every column for every row of a table in an RDBMS must have some representation of a value, so the only option to remove a value is to set it to NULL to indicate the absence of a value.

Not every column allows null values, but we know of at least one in our database that does: the MiddleName column in the author table. If we wanted to set the MiddleName for the first author to NULL, we would write an UPDATE statement like this:

```
UPDATE author
SET MiddleName = NULL
WHERE AuthorID = 1;
```

These are our options for updating values using a basic UPDATE statement, but now let's look at how we will use an update statement when we need to query multiple tables as part of our filtering condition.

17.1.4 Updating values with a multi-table query

Recall in chapter 8 we noted that a *predicate* is any part of our SQL statement that evaluates whether something is true, false, or unknown. Since then, we have used predicates for filtering conditions, which have included conditions in the FROM, WHERE, and HAVING clauses.

As you work with SQL, you will likely need to write an UPDATE statement that requires filtering with a predicate that uses more than one table. This can be tricky because we can only update values in one table in an UPDATE statement, and we want to make sure we do this correctly. To do this, we will need a FROM clause to identify the filtering conditions, and we need to make sure we put this in the correct place in our SQL statement.

Unfortunately, there are different places to put this predicate in an UPDATE statement, depending on which RDBMS you are using. While many uses of keywords are standardized for every RDBMS, there exists no standard for updating values in a query with multiple tables.

NOTE Although this section will contain examples for updating values in some commonly used RDBMSs outside of MySQL, this will not be a comprehensive list of examples. Be sure to check the documentation of whatever RDBMS you might find yourself using to confirm the correct syntax for these kinds of UPDATE statements.

Suppose we wanted to update the price for any titles from a particular author in our sqlnovel database in MySQL. We know the AuthorID, which is 12, but we do not have the TitleName or TitleID of any particular title. We also know we want to update the price to \$8.95. In order to complete this update, we will need to join at least two tables to complete the update query.

As a refresher, the relationship between the title and author table is established through the titleauthor table. Rows are uniquely identified in title by TitleID, and in author by AuthorID. The titleauthor table contains columns for both TitleID and AuthorID in a many-to-many relationship since any author could potentially write more than one title, and any title could have more than one author.

Before we dive into the UPDATE statement, let's first start with a SELECT query to see the current value we intend to update. If we wanted to write a query that shows the Price for any titles by the author with AuthorID of 12, we might write a query using joins in the FROM clause like this:

```

SELECT title.Price
FROM title
INNER JOIN titleauthor
    ON title.TitleID = titleauthor.TitleID
INNER JOIN author
    ON titleauthor.AuthorID = author.AuthorID
WHERE author.AuthorID = 12;

```

This query returns the value we intend to change, but we can make this a bit more readable using table aliases, as we discussed in chapter 8. Let's use some table aliases to reduce the wordiness of our query:

```

SELECT t.Price
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
INNER JOIN author a
    ON ta.AuthorID = a.AuthorID
WHERE a.AuthorID = 12;

```

There—that's a bit easier to read. Savvy readers may also note that we really don't need the author table in this query, since the AuthorID value is also included in the titleauthor table. We can make our query simpler by removing any reference to the author table and filtering on AuthorID in the titleauthor table:

```

SELECT t.Price
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;

```

Now that we have the foundation of a SELECT statement, we can easily change this into our desired UPDATE statement by removing the SELECT clause in our query and replacing the FROM keyword with UPDATE. Then, after the UPDATE, we can use SET to set the values before the WHERE clause. Because we have established table aliases in the previous FROM clause, we can use the same table aliases in our query to indicate which table in our query is having data updated. Here is what our UPDATE statement will look like:

```
UPDATE title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
SET t.Price = 8.95
WHERE ta.AuthorID = 12;
```

This can be a bit confusing, since we have split the predicate into two separate parts of our query, with the joining of tables in the update before the SET, and the rest of the filtering after the SET in the WHERE clause. Again, this method of updating is specific to MySQL.

NOTE The following examples are for some common RDBMSs other than MySQL. These examples are provided to show differences and are not intended to be executed in MySQL. If you attempt to execute them, they will result in failed queries that error.

If our database were using PostgreSQL, it would be different in that we would still indicate the table to be updated and any alias in the UPDATE portion of our query, but we would use the FROM clause to join any additional tables. (This query will not work in MySQL.)

```
UPDATE title t
SET t.Price = 8.95
FROM titleauthor ta
WHERE t.TitleID = ta.TitleID
    AND ta.AuthorID = 12;
```

If our database was in a SQL Server instance, the logic might make a bit more sense compared to the SELECT query. In SQL Server, we would simply replace the SELECT with the UPDATE and SET parts, and then leave the FROM and WHERE clauses unchanged. (This query will also not work in MySQL.)

```
UPDATE title t
SET t.Price = 8.95
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

The point is that, although you can use joins for filtering in an UPDATE statement, each RDBMS will have its own particular syntax for these kinds of queries. Unfortunately, this is one of those rare parts of learning SQL where you may need to venture beyond this book to learn the specific syntax of the RDBMS you are using.

Now, let's move on to discussing how to delete data in SQL.

17.2 Deleting rows

When we delete data using SQL, we are effectively doing the opposite of what an INSERT statement does. While INSERT adds one or more rows to a table, DELETE removes one or more rows. And just like INSERT and UPDATE statements, we can only change data in one table per DELETE statement.

17.2.1 Deleting one or more rows

As we have done so many times, let's start with a verbal declaration. Suppose we want to delete the row from the title table where the TitleID = 110. We might start with a verbal declaration like this.

"I would like to delete any rows from the title table where the TitleID is 110."

Our SQL statement will be very close to this, using a new keyword: DELETE

```
DELETE
FROM title
WHERE TitleID = 110;
```

Like INSERT and UPDATE statements, we are starting our statement with a data manipulation keyword, which in this case is DELETE. Next, we identify the table where we intend to delete data. Finally, we indicate the filtering to be used so we only delete the intended rows.

WARNING Just like UPDATE statements, the WHERE clause is not required but is perhaps the most critical part of our query. If you do not write *and execute* this filtering condition, then you will delete *every row in the table*. As you can imagine, this can be a catastrophic result for your query. Because these changes are in real-time, always take the utmost care to write *and execute* the filter to delete only the intended rows.

Of course, we can use variables to make repeatable SQL statements for a query like this. Here is how we can do this for the preceding query:

```
SET @TitleID = 110;

DELETE
FROM title
WHERE TitleID = @TitleID;
```

TIP Some RDBMSs allow the omission of the FROM keyword in DELETE statements like these, but it's best to use it anyway. Although it is preferable to reduce the wordiness of queries when possible, omitting one word doesn't reduce the wordiness by a significant amount. Moreover, if you had to migrate your query from one RDBMS to another, omitting the word FROM could result in a query that doesn't work after such a migration.

17.2.2 Deleting a row with a multi-table query

In section 17.1.4 we discussed how the syntax for an UPDATE statement that joins multiple tables in the predicate can vary depending on which RDBMS you are using. Unfortunately, the syntax for DELETE statement can also vary from one RDBMS to another. The good news is that the syntax doesn't vary quite as much, as MySQL, SQL Server, and MariaDB all share the same syntax.

Even better news is that this syntax is remarkably similar to a SELECT statement. Recall how in section 17.1.4 we created this SELECT statement before writing SQL for an UPDATE statement for a title that related to AuthorID 12:

```
SELECT t.Price
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

If we wanted to delete the row in title instead of update the value for Price, we would simply replace the SELECT clause with a DELETE that noted the table to be deleted, like this:

```
DELETE t
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

Again, this syntax works in MySQL and a few other RDBMSs, but not all. Please refer to the documentation of the particular RDBMS you are using for the correct syntax to delete rows using a predicate that joins multiple tables.

17.2.3 Deleting all rows in a table

As noted in the warning in section 17.2.1, if a DELETE statement is executed without a WHERE clause, it can result in removing all the rows from a table. I say "can" because, as noted in chapter 16, databases are often designed with certain kinds of constraints including those that allow or prevent certain kinds of values from being inserted into a table. Those same kinds of constraints can allow or prevent rows from being deleted as well.

If we do not have these kinds of constraints, and the tables in our database currently do not, we can remove all the rows in a table by executing a DELETE statement that has no filtering condition. Obviously, we don't want to delete all the rows from the tables in our database, but for the sake of practice, we can execute this on the myfirstquery table we used in chapter 2 with minimal impact on any of our future exercises. Here is what that query would look like:

```
DELETE
FROM myfirstquery;
```

As expected, executing this query will result in the deletion of all rows in the myfirstquery table. Additionally, we have another more efficient way to remove all rows from a table that is available in nearly every RDBMS: TRUNCATE TABLE.

Unlike removing all rows with a DELETE statement, which scans every row in a table and deletes them one at a time, TRUNCATE TABLE is designed for speed by deleting rows without scanning them individually. The syntax for TRUNCATE TABLE is very simple. If we wanted to truncate the myfirstquery table, our SQL statement would look like this:

```
TRUNCATE TABLE myfirstquery;
```

The TRUNCATE TABLE command has a lot less flexibility than a DELETE statement, as it is designed for the singular purpose of quickly removing all rows from a table. It cannot be used to remove one row or some rows. Also, the same issues with constraints that can prevent rows from being deleted with a DELETE statement can also prevent TRUNCATE TABLE from being executed successfully.

TRY IT NOW

Use the previous DELETE and TRUNCATE TABLE statements to remove all the rows from the myfirstquery table. If you want to test deleting multiple rows, try executing one or more INSERT statements to add rows to the myfirstquery table.

17.3 One big tip for data manipulation

In section 17.1.4, we wrote a SELECT statement to find the data we were going to update. Although we did this to use the basic structure of the SELECT statement to compare it to our UPDATE statement, we just as easily could have used this SELECT statement to validate which rows were going to be updated.

Two separate warning sections in this chapter caution against the accidental updating or deleting of all data in a table. There could be even more warnings about updating the wrong data through incorrect logic for filtering contained in the predicates of UPDATE and DELETE statements. Remember, all changes occur in real-time, so the utmost caution should be used when updating and deleting data.

One significant and common way to exercise caution is by writing and executing a SELECT statement that uses the same logic as contained in an UPDATE or DELETE statement. Doing this allows you to see what data is going to be affected before you execute your data manipulation statement, and I can tell you from personal experience that selecting data through a query first in this way has saved me and countless others from catastrophic results.

Although there is no way to undo an executed query, you can always proactively review the affected rows using a SELECT. I highly recommend selecting data with your filtering conditions, no matter how experienced you get at writing SQL.

17.4 Lab

1. Using what you learned in chapter 16, write and execute a query to add yourself and your address to the customer table. Use the value 22 for CustomerID.
(HINT: You can drag and drop column names from the Navigator panel in the Workbench.)
2. Write and execute a query to update the address of the row you just added to a previous address you previously resided (or any other address).
3. Write and execute a query to delete the row you just inserted and updated in the customer table, but use a SELECT first to confirm your SQL statement will produce the correct results.

17.5 Lab answers

1. Your query to insert a row in the customer table might look something like this:

```

INSERT customer (
    CustomerID,
    FirstName,
    LastName,
    Address,
    City,
    State,
    Zip,
    Country
)
VALUES (
    22,
    'Jeff',
    'Iannucci',
    '1600 Pennsylvania Ave NW',
    'Washington',
    'DC',
    '20500',
    'USA'
);

```

2. Your query to update your address in the customer table might look something like this:

```

UPDATE customer
SET Address = '1700 W Washington St',
    City = 'Phoenix',
    State = 'AZ',
    Zip = '85007'
WHERE CustomerID = 22;

```

3. Your query to delete your row from the customer table might look something like this:

```

DELETE
FROM customer
WHERE CustomerID = 22;

```

18

Storing data in tables

In chapters 16 and 17 we started creating and manipulating data, and in this chapter, we're going to examine the ways we can create and manipulate the tables themselves. In many ways, we will be getting to the core of SQL language and its usage, because how the data is stored in tables is at the very heart of any RDBMS. How this data is stored is one of the most important decisions – perhaps the most important decision – that is made in any database.

Don't worry, this won't be overly technical. It will still be easy enough for anyone to understand, and since you have been querying data with SQL statements that often mirror the English language for a while now, I'm confident you will find the concepts and commands easy to understand. In fact, this chapter should help reinforce your understanding of things like primary keys and data types.

Let's get started and create a new table in our database!

18.1 Creating a table

As you'll soon see, creating a table in SQL can be very simple. However, first we must consider a few things about the table before we write the SQL that creates the table.

18.1.1 Considerations before creating a table

The first step in creating a table is to answer three basic questions about the table:

1. What is the name of the table?
2. What are the names of the columns that will be included in the table?
3. What are the data types of those columns?

Some thought will need to go into these names and data types, especially as they relate to existing tables in the database. We want the names to be explanatory, but we also can't use a table name if there is already a table with that name.

However, we can reuse column names from other tables, as we've seen through the `sqlnovel` database with `OrderID`, `TitleID`, etc. appearing in more than one table. But we only want to do that if there is some relationship between similarly named columns.

TIP This idea of having similar names is one of the reasons why names such as "OrderID" and "TitleID" have been used in our database. You may one day find yourself working with a database that has many tables with columns named "ID" or "Name", which can be a bit confusing due to ambiguity. As you create tables, try to make column names meaningful, clear, obvious, and easy to understand for anyone else who may have to query the data.

As an exercise, we want to create a table for categories of the titles. Although we could avoid creating a new table and simply add a "Category" column in the title table for a string of characters such as "Mystery" or "Romance," we know from our discussion of table relationships in chapter 8 that this isn't the best approach in an RDBMS. We will add a column to the title table later in the chapter, but it will be a column that relates to the primary key of our new table.

So, back to our new table of categories, we only need to have two columns: an ID column to serve as the primary key, and a column to define the name of each category. We can keep the naming of the table and columns consistent with the rest of our database by naming the table "category" and the columns "CategoryID" and "CategoryName" respectively.

The ID column of a table is often defined as an integer data type, which is known as `int`. The `int` data type allows us to use a unique number for each row, usually up to a number in the billions. I don't think we are going to have billions of categories, so an integer data type should work well for our `CategoryID`. There are some other integer data types we could use, such as `tinyint`, `smallint`, `mediumint`, and `bigint`, but since they aren't included in every RDBMS, we will use the universal `int` data type.

Columns with name values of varying length like `CategoryName` are typically defined as a variable character type, known as `varchar`. This data type is used because it accounts for the fact that not every value will be the same length, which is to say the number of characters. By using this data type, we allow our data to be stored more efficiently than if we had used a `char` (character) data type, since that data type stores the data using the entire defined length.

Although defining the maximum length isn't required for `varchar` data types, we typically want to do this to avoid using an inefficient default value, which will vary depending on our RDBMS. The values in our `CategoryName` column won't be more than 20 characters, so we will define the data type as `varchar(20)`.

NOTE As you converse with others about SQL, you will discover there is no common way to pronounce “varchar”. Pronunciations vary from the literal “var-char” to “var-kar” to “vair-kair.” The last option is probably the most likely to be correct, since the vowels match the pronunciation of the first syllables in “variable character,” but I’ve found it’s also the least likely to be used. Try not to get too confused by this. As the French say, *vive la différence*.

18.1.2 Creating a table

Now that we have defined our table name, column names, and column data types, let’s say in English what we intend to do.

“I would like to create a table named ‘category’. I would like the table to have a column named ‘CategoryID’ that is an int data type. I would like the table to also have a column named ‘CategoryName’ that is a varchar(20) data type.”

After all that, here is what our SQL will look like to create our category table:

```
CREATE TABLE category (
    CategoryID int,
    CategoryName varchar(20)
);
```

Notice how after we have defined the name of the table, we have included all columns with a comma separating the names and data types of each column. This should seem a bit intuitive now after seeing how commas are used to separate columns in SELECT and ORDER BY clauses. But also notice how the columns are enclosed in parentheses. Omitting the parentheses when using CREATE TABLE can be a common mistake for beginners, so always remember to include them when creating a table.

Executing the preceding SQL will not return any results in the Results panel, but in the Output panel you should see a green circle with a white checkmark and the Message “0 rows(s) affected.” Although there isn’t a lot of detail there, that tells you the table was created successfully.

Creating a table is very simple, although this is just a basic starting point for creating a table. As you’ll see later in this chapter and subsequent chapters, the CREATE TABLE statement allows us quite a few options for adding more properties to our table.

For now, we will move on to the next step of adding data to our table, using what we learned about the INSERT keyword in chapter 16.

18.1.3 Adding values to an empty table

Our customer table is empty, so next, we want to insert the CategoryID and CategoryName values for the following categories:

1. Romance
2. Humor
3. Mystery

4. Fantasy
5. Science Fiction

In chapter 16, we discussed how to add multiple values to a table using the `INSERT` and `VALUES` keywords. Let's use similar logic to insert these listed values into our new category table:

```
INSERT INTO category (CategoryID, CategoryName)
VALUES
(1, 'Romance'),
(2, 'Humor'),
(3, 'Mystery'),
(4, 'Fantasy'),
(5, 'Science Fiction');
```

Executing the preceding SQL will also not return anything in our Results panel, but if the execution is successful, we should see "5 row(s) affected" in the Message column of the Output panel. This indicates we added five rows to our category table, which was our intention.

TRY IT NOW

If you haven't created and populated the category table yet, execute the SQL in 18.1.2 and 18.1.3 to do so. We will be using this table not only in this chapter but also in subsequent chapters.

We can verify that we have added the desired values to the category table by running a simple `SELECT` query to see the values in the table, with the results shown in figure 18.1:

```
SELECT
    CategoryID,
    CategoryName
FROM category;
```

CategoryID	CategoryName
1	Romance
2	Humor
3	Mystery
4	Fantasy
5	Science Fiction

Figure 18.1 All rows in the new category table

We have created and populated the table for our categories, so the next step will be to relate these categories to our titles in the title table.

18.2 Altering a table

Our next step in adding a category for each title will be to add a new column to the title table that relates to the values in our new category table. Specifically, we want to relate the CategoryID values from the title table to those in the category table. We will do this by adding a CategoryID column to our title table.

18.2.1 Adding a column to a table

Just as we had to consider three questions before creating a table, we need to consider three similar questions before adding a column. Let's revisit the list:

1. What is the name of the table we are adding the new column to?
2. What are the names of the columns that will be added to the table?
3. What are the data types of those columns?

We know the answer to question 1 is our title table, and because we want consistency in names and data types, the answers to question 2 and question 3 will be the same as the ID value in the category table: CategoryID and int, respectively.

To make these and other changes to a table in SQL, we are going to use a new command: ALTER TABLE. Although the syntax will be similar to CREATE TABLE, it will not require us to use parentheses. Here is what our SQL to add the column to the title table will be:

```
ALTER TABLE title
ADD CategoryID int;
```

As with our CREATE TABLE statement in section 18.1.2, we will not have any results from this query. The success of this query will only be noted in the Output panel with a white checkmark in a green circle and the message "0 row(s) affected."

Let's run a quick query to validate that the column was created, noting the results in figure 18.2 that show the column was created after all the other columns:

```
SELECT *
FROM title;
```

TitleID	TitleName	Price	Advance	Royalty	PublicationDate	CategoryID
101	Pride and Predicates	9.95	5000.00	15.00	2015-04-30 00:00:00	NULL
102	The Join Luck Club	9.95	6000.00	12.00	2016-02-06 00:00:00	NULL
103	Catcher in the Try	8.95	5000.00	10.00	2017-04-03 00:00:00	NULL
104	Anne of Fact Tables	12.95	10000.00	15.00	2018-01-12 00:00:00	NULL
105	The DateTime Machine	7.95	5500.00	15.00	2019-02-04 00:00:00	NULL
106	The Great GroupBy	10.95	0.00	20.00	2019-12-23 00:00:00	NULL
107	The Call of the While	8.95	2500.00	15.00	2020-03-14 00:00:00	NULL
108	The Sum Also Rises	7.95	5000.00	12.00	2021-11-12 00:00:00	NULL
109	David Emptyfield	9.95	0.00	10.00	2022-01-16 00:00:00	NULL
110	Red Badge of Cursors	7.95	0.00	15.00	2022-03-29 00:00:00	NULL
111	Of Mice and Metadata	8.95	0.00	12.00	2022-05-17 00:00:00	NULL
112	A Table of Two Cities	9.95	0.00	15.00	2022-08-07 00:00:00	NULL

Figure 18.2 The title table with the new CategoryID column at the far right

Our new column doesn't have any values yet, so the results of our query show NULL for every row. To add the values, we will use UPDATE statements. Although we used INSERT to add values to our category table, INSERT adds an entirely new row to a table. We don't want to do that here; we just want to add a value for a single column, which is what UPDATE allows us to do.

Let's add these values for all rows, based on the category of each title. In case you didn't memorize all the CategoryID and CategoryName values or you don't feel like flipping back a few pages, the following SQL has comments to help remind you:

```
/* 1 - Romance */
UPDATE title
SET CategoryID = 1
WHERE TitleID IN (101, 104);

/* 2 - Humor */
UPDATE title
SET CategoryID = 2
WHERE TitleID IN (106, 109);

/* 3 - Mystery */
UPDATE title
SET CategoryID = 3
WHERE TitleID IN (102, 103, 110);

/* 4 - Fantasy */
UPDATE title
SET CategoryID = 4
WHERE TitleID IN (107, 112);

/* 5 - Science Fiction */
UPDATE title
SET CategoryID = 5
WHERE TitleID IN (105, 108, 111);
```

After executing all the preceding update statements, we should see values for the CategoryID column of all rows. Let's execute our query to return all rows of the title table, verifying in figure 18.3 that all values for CategoryID are now populated:

```
SELECT *
FROM title;
```

TitleID	TitleName	Price	Advance	Royalty	PublicationDate	CategoryID
101	Pride and Predicates	9.95	5000.00	15.00	2015-04-30 00:00:00	1
102	The Join Luck Club	9.95	6000.00	12.00	2016-02-06 00:00:00	3
103	Catcher in the Try	8.95	5000.00	10.00	2017-04-03 00:00:00	3
104	Anne of Fact Tables	12.95	10000.00	15.00	2018-01-12 00:00:00	1
105	The DateTime Machine	7.95	5500.00	15.00	2019-02-04 00:00:00	5
106	The Great GroupBy	10.95	0.00	20.00	2019-12-23 00:00:00	2
107	The Call of the While	8.95	2500.00	15.00	2020-03-14 00:00:00	4
108	The Sum Also Rises	7.95	5000.00	12.00	2021-11-12 00:00:00	5
109	David Emptyfield	9.95	0.00	10.00	2022-01-16 00:00:00	2
110	Red Badge of Cursors	7.95	0.00	15.00	2022-03-29 00:00:00	3
111	Of Mice and Metadata	8.95	0.00	12.00	2022-05-17 00:00:00	5
112	A Table of Two Cities	9.95	0.00	15.00	2022-08-07 00:00:00	4

Figure 18.3 The title table with the CategoryID populated with values for all rows

We have confirmed that we have added the values for CategoryID, but those are just ID values that don't tell us directly what the category names are for each title. Let's show the CategoryName for each TitleName by writing a query that relates the title and category tables to each other by CategoryID. This relationship will be established with an INNER JOIN, like those we discussed in chapter 8. The results in figure 18.4 confirm the category for each title, ordered by TitleID:

```

SELECT
    t.TitleID,
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID
ORDER BY t.TitleID;

```

TitleID	TitleName	CategoryName
101	Pride and Predicates	Romance
102	The Join Luck Club	Mystery
103	Catcher in the Try	Mystery
104	Anne of Fact Tables	Romance
105	The DateTime Machine	Science Fiction
106	The Great GroupBy	Humor
107	The Call of the While	Fantasy
108	The Sum Also Rises	Science Fiction
109	David Emptyfield	Humor
110	Red Badge of Cursors	Mystery
111	Of Mice and Metadata	Science Fiction
112	A Table of Two Cities	Fantasy

Figure 18.4 The CategoryName for every TitleName in the title table, as related by CategoryID

Adding a category for each title can be relatively simple, but as with creating a table, there are a few things to consider before simply adding new columns to tables.

18.2.2 Considerations before adding a column

When we add a column to a table, one of the items that may bother a SQL novice is that the new column will be added at the end of all the other columns. It's always going to be added at the end, although some RDBMSs such as MySQL will allow you to change the order of the columns. However, this is generally discouraged on tables that already have data.

There are two main reasons for adding columns only at the end of all other columns. The first reason is that adding a new column before any columns will create a lot of extra activity from rearranging the data, using more resources than simply adding the column at the end. When we add the column at the end, we minimize the amount of activity and resources that will be required.

The second reason is that it is generally unnecessary to position columns in a certain order for a table, as you've already seen columns can be ordered to be displayed however desired in our SQL queries.

There is one exception to adding a column at the end though, and that is if you are creating a new column to be used in the primary key of a table. We generally want to have those columns first because the primary key will often define the way the data will be ordered in a table. Although we've referred to primary keys since chapter 8, let's take a deeper look at how we create and use them.

18.3 Primary keys

This book has been designed to get you gradually up to speed with writing SQL queries, so we haven't discussed primary keys much since chapter 8. However, as we are talking about designing our own tables and relating data that we created, it's time to revisit the topic of primary keys.

18.3.1 Considerations for primary keys

Primary keys are the backbone of any RDBMS, as they ensure that data in tables is relatable. In order for them to do this, we have to adhere to a few rules:

- **Primary keys must be unique.** This is the first and most important rule. If they aren't unique, then we can never know which row is the correct one in a relationship. Consider if we had assigned a value of 1 for the CategoryID of each of the five rows in our new category table. If all five rows had the same value, we couldn't possibly determine the correct CategoryName for any relationship where the CategoryID is 1.
- **Every row must have a value for the primary key.** As we've previously seen, we can't join a value of NULL to any value, including other values of NULL. Because of this, if any row in a table has NULL for a primary key value, then that row can never be related to any other data.
- **Primary key values can never change.** We use these primary key values to relate to other tables, much like we used the CategoryID values to relate to the title table. If we changed the CategoryID values in the category table in any way, such as adding 10 to the existing value, the values in the two tables would not match and we would no longer have the relationship between the two tables.

Adhering to these three rules will allow relationships that use primary keys to maintain *referential integrity*, which means that any value from one table that refers to the primary key value in another table will always refer to the same value in the table with the primary key. Although the values for other columns in our table can change, such as changing a CategoryName from "Mystery" to "Mystery Thriller", the values for the primary key can never change.

18.3.2 Adding a primary key

Now that we've covered the rules for any primary key, we can create one for our category table. To do this, we can use the ALTER TABLE statement we've already used to add a column, with some slight differences:

```
ALTER TABLE category
ADD CONSTRAINT PRIMARY KEY (CategoryID);
```

Using ALTER TABLE, there are two notable exceptions between adding a column and adding a primary key. First, we used the word CONSTRAINT when noting that we were adding a primary key. We haven't explicitly said it before now, but a primary key is a kind of *constraint*. This term means it is an object designed to specify a rule of some sort, and as such, we can't break the rules of the constraint once it is created. For a primary key, the rules outlined in section 18.3.1 would now be adhered to concerning the category table.

Although it isn't required, it is common to name a primary key with a logical name. The reason for this is you may be creating other objects including different constraints on your table, and you will want the names of these objects to indicate some bit of information about the purpose of each object. Just as we give tables obvious names for their data, such as "Customer" or "Author", we want to give our primary keys obvious names.

Moreover, as you'll see later in this chapter, having a name for a primary key, or any other kind of constraint, makes it much easier to manipulate if you find you have to change or even delete the constraint.

A common way to name primary keys is to use the naming convention of "PK_" as a prefix and then the table name. Even if we knew nothing about a particular database, if we saw any reference to an object named "PK_category" it would be almost universally understood to be the primary key constraint of a table named "category". Given this, it would be preferable to create our primary key with a name, like this:

```
ALTER TABLE category
ADD CONSTRAINT PK_category PRIMARY KEY (CategoryID);
```

TRY IT NOW

Create the primary key constraint for the category table with the preceding SQL statement.

Although we are adding the primary key to an existing table, most of the time you will add the primary key at the time you create the table. This is easily accomplished by defining the primary key in the CREATE TABLE statement after the columns have been defined, almost as if this were another column. For example, we could have created our category table with the primary key defined with the following SQL:

```
CREATE TABLE category (
    CategoryID int,
    CategoryName varchar(20),
    CONSTRAINT PK_category PRIMARY KEY (CategoryID)
);
```

TIP While it isn't necessary to create a primary key for every table in every database, a primary key definitely belongs on any table that has, for lack of a better phrase, a group of unique entities. Tables with rows that represent unique entities such as products, orders, and customers should always have a primary key established to ensure those rows are unique.

One final note about primary keys is that they can consist of more than one column. For example, our orderitem table should have a primary key that includes both the OrderID and the ItemID because, together, these will form a unique key. There will be multiple rows with the same OrderID, but each of those rows should have a unique ItemID for every row that matches that OrderID.

To create a primary key with multiple columns on a table such as the order item table, we would only need to use a comma separator when specifying the columns, like this:

```
ALTER TABLE orderitem
ADD CONSTRAINT PK_orderitem PRIMARY KEY (OrderID, OrderItem);
```

Using primary keys allows us to ensure we can identify each row in a table as unique. Because these rows are often referred to by other tables in relationships, we can explicitly define these relationships with another kind of constraint: the foreign key.

18.4 Foreign keys and constraints

Foreign keys are established to enforce the rules of any relationship between two tables, with respect to a common column. In any foreign key relationship, there is a *parent table* where the source of the key values originates, and the *child table* where there are values that refer to the parent table. Put simply, we create a foreign key on the child table to enforce the rule that the values in the child table must exist in the parent table.

Let's look at a diagram of our sqlnovel database to understand foreign keys a little more.

18.4.1 Data diagrams

Data diagrams can help us better understand the relationships between tables in our database by visually showing us these relationships. In figure 18.5 we can see a data diagram of all the tables in the sqlnovel database, including our new category table.

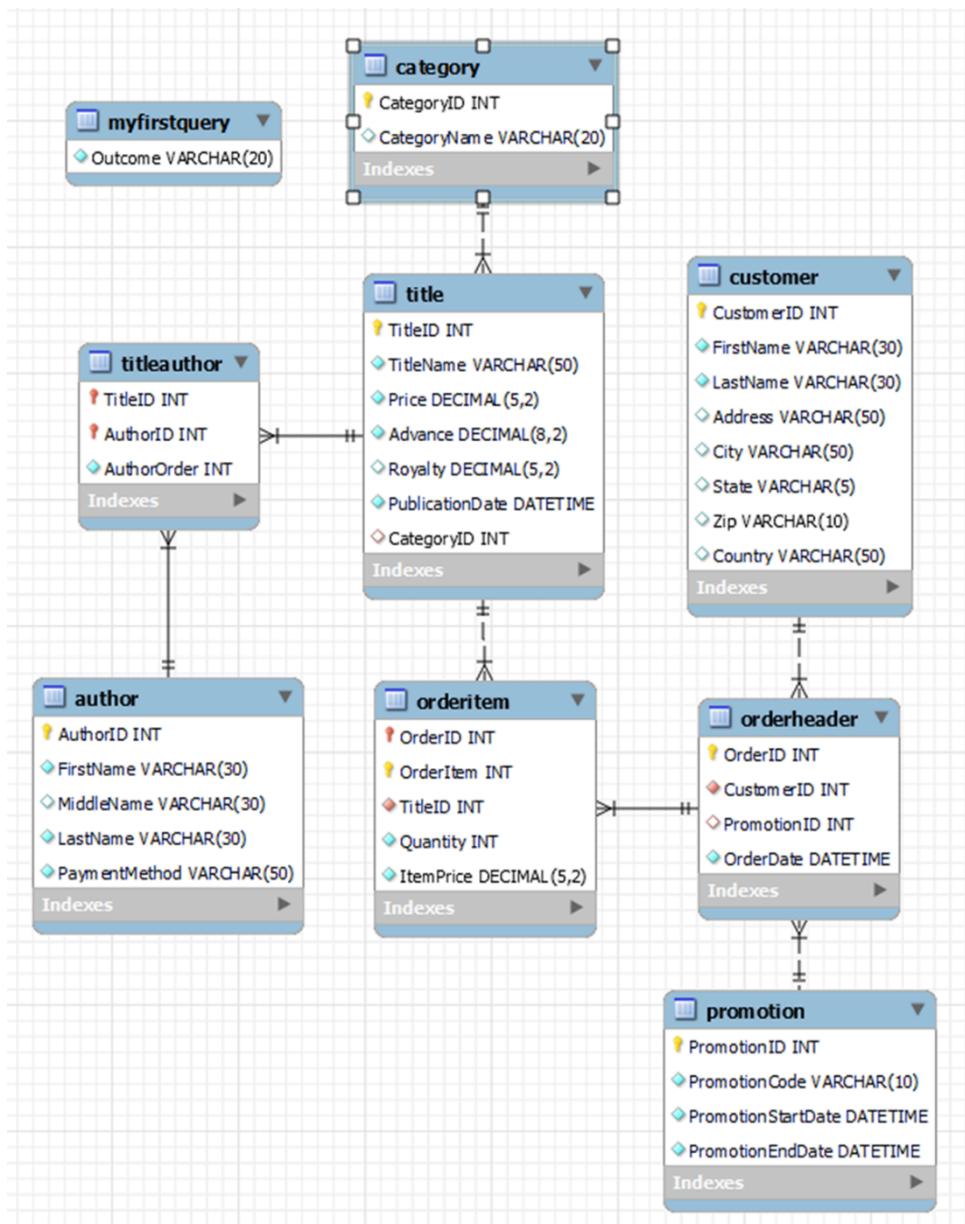


Figure 18.5 A data diagram that includes all tables in the sqnolovel database and their relationships to other tables

Each box in this diagram represents a table, and within each box is a list of all columns and data types for that table. What we want to focus on here is the lines between the boxes, which represent the relationships. If a line exists between two tables, then a foreign key from one table to another exists.

Although the data diagram in figure 18.5 doesn't show the specific columns involved in the relationships, we can find those out by examining the tables more closely. In fact, if you follow through and execute the prescribed exercises throughout this chapter and in the lab, at the end of the lab you will see how to create your own data diagram.

One other thing to note is that the myfirstquery table has no lines connecting it to other tables. That table has no relationship to any other tables in our database, as it was used only to help get started with your first query in chapter 2. You've come a long way since then!

18.4.2 Adding a foreign key constraint

Since a foreign key constraint is another type of constraint, we will use an ALTER TABLE statement that is similar to the one we used to create the primary key constraint. As with the primary key, we want to give a logical name for our foreign key constraint. One common way is to use a name with a prefix of "FK_", then the name of the child table, then another underscore, and then the name of the parent table. This ensures the name of our foreign key constraints will be unique throughout the database.

Using this naming convention, we can create our foreign key constraint with the following statement:

```
ALTER TABLE title
ADD CONSTRAINT FK_title_category
FOREIGN KEY (CategoryID) REFERENCES category(CategoryID);
```

TRY IT NOW

Create the foreign key constraint on the title table with the preceding SQL statement.

This is a bit different than the statement we used to create the primary key constraint because we are creating a key relationship between two columns in different tables. The first column, after the keywords FOREIGN KEY, indicates the column on the child table that will reference another column in the parent table. This is why the keyword REFERENCES is used.

We want to be careful when we create constraints – primary key, foreign key, or otherwise – on tables that already have data because, if the current values don't meet the rules of our constraint, we will have an error returned. For this reason, it is best to create constraints on tables when the table is first created and before any rows of data have been inserted.

TIP Although we have used common conventions to name our constraints, when working outside of our sqlnovel database, you should consider if the database you are working with already has any defined naming conventions that are being used. If it does, create your objects with the existing naming conventions so your object names are consistent with other objects in the database.

18.5 Deleting a table, column, or constraint

Although we want to keep the objects we have created in this chapter, if we want to undo the work we have done, we can do this with different statements that use the `DROP` keyword.

WARNING The SQL provided in this section is for informational purposes only. We don't really want to drop the category table or any of its related columns and constraints, as we will be using this data throughout the remainder of the book. If you decide that you want to practice dropping these objects despite this warning, then you will want to go back through the previous parts of this chapter to re-create them all.

With that warning out of the way, here's how we could remove the objects we have created in this chapter.

18.5.1 Deleting a constraint

Just like adding a constraint, deleting any constraint is going to use the `ALTER TABLE` statement. Because we are only dropping our constraint and aren't defining anything, we only need the names of the table and the constraint that we are dropping. In this case, we would use the following SQL:

```
ALTER TABLE title
DROP FOREIGN KEY FK_title_category;
```

If we wanted to delete the `PK_category` primary key of our category table, there are actually two ways we could do this. The first would be to remove it as a constraint, like this:

```
ALTER TABLE category
DROP CONSTRAINT PK_category;
```

However, since this constraint is a special kind of constraint, we could also use `ALTER TABLE` to say we want to remove the primary key:

```
ALTER TABLE category
DROP PRIMARY KEY;
```

Note that we don't need to specify the name of the primary key in the statement. This is because the table can only have one primary key.

18.5.2 Deleting a column

In this chapter, we also created a column on the title table. To remove that column, we would use ALTER TABLE and DROP, like this:

```
ALTER TABLE title
DROP COLUMN CategoryID;
```

As with dropping constraints, we typically don't need more than the names of the table and column to remove a column.

18.5.3 Deleting a table

Deleting a table involves the least SQL of any of our object removal scripts. We just say DROP TABLE with the table name:

```
DROP TABLE category;
```

NOTE If we wanted to remove all these objects, we would need to do it in the order we used here in section 18.5, which is to remove the constraints first. If we wanted to drop the table or column first, most RDBMSs including MySQL would return an error that the object cannot be dropped because of the existence of these constraints.

That's enough discussion of removing objects. Let's start the lab where we can practice creating some constraints, such as primary keys and foreign keys.

18.6 Lab

1. Believe it or not, the sqlnovel database is missing a few primary keys. Using the data diagram in section 18.4.1 and the naming conventions noted in section 18.3.2, write and execute SQL statements to add the primary key constraints for the following tables:
 - author
 - customer
 - orderheader
 - promotion
 - title
 - titleauthor

2. The orderitem table is not included in the preceding list. What happens if you try to create a primary key constraint on the OrderID and OrderItem columns? How can you resolve this?
3. As it turns out, the sqlnovel database is also missing a few foreign key constraints. Using the data diagram in section 18.4.1 and the naming conventions noted in 18.4.2, write and execute SQL statements to add the foreign key constraints for the following tables and columns:
 - The CustomerID column on the orderheader table
 - The PromotionID column on the orderheader table
 - The OrderID column on the orderitem table
 - The TitleID column on the orderitem table
 - The TitleID column on the titleauthor table
 - The AuthorID column on the titleauthor table
4. If you've successfully completed all the preceding tasks, now is a chance to enjoy your work by creating a data diagram. In the SQL Workbench, go to the top menu and select Database > Reverse Engineer. Follow through the menu choices by selecting Next at all screens, and be sure to select the checkbox for sqlnovel on the Select Schemas screen. When you are done, you should have a data diagram of the sql novel database.

18.7 Lab answers

1. The primary key constraints for these tables can be created with the following SQL statements:

```
ALTER TABLE author ADD CONSTRAINT PK_author PRIMARY KEY (AuthorID);
ALTER TABLE customer ADD CONSTRAINT PK_customer PRIMARY KEY (CustomerID);
ALTER TABLE orderheader ADD CONSTRAINT PK_orderheader PRIMARY KEY (OrderID);
ALTER TABLE promotion ADD CONSTRAINT PK_promotion PRIMARY KEY (PromotionID);
ALTER TABLE title ADD CONSTRAINT PK_title PRIMARY KEY (TitleID);
ALTER TABLE titleauthor ADD CONSTRAINT PK_titleauthor PRIMARY KEY (TitleID, AuthorID);
```

2. The statement to create the primary key constraint on orderitem would look like this:

```
ALTER TABLE orderitem ADD CONSTRAINT PK_orderitem PRIMARY KEY (OrderID, ItemID);
```

However, if you execute this statement, you will see in the Output window an error that says, "Error Code: 1062. Duplicate entry '1022-1' for key 'orderitem.PRIMARY'."

This error indicates there is a data inconsistency in what would be our primary key, and it tells you where that error is. The error is for OrderID 1022 and OrderItem 1.

We can see the problem in figure 18.6 with the following query, which should return one row but instead returns two:

```
SELECT *
FROM orderitem
WHERE OrderID = 1022
    AND OrderItem = 1;
```

OrderID	OrderItem	TitleID	Quantity	ItemPrice
1022	1	101	1	7.95
1022	1	103	1	6.95

Figure 18.6 The two rows that prevent the primary key from being created on the orderitem table

There are two rows that would result in duplicate values for our primary key, which is not allowed. All key values must be unique. Fortunately, these rows do not appear to be actual duplicates, as they have different TitleID values. We can safely correct this data error with an UPDATE statement, changing the OrderItem value from 1 to 2 for one of the rows, like this:

```
UPDATE orderitem
SET OrderItem = 2
WHERE OrderID = 1022
    AND OrderItem = 1
    AND TitleID = 103;
```

After executing the preceding UPDATE statement, we should now be able to create the primary key as desired on the orderitem table. Go ahead and create that primary key constraint.

3. The foreign key constraints for these tables and columns can be created with the following SQL statements:

```
ALTER TABLE orderheader ADD CONSTRAINT FK_orderheader_customer FOREIGN KEY (CustomerID)
REFERENCES customer(CustomerID);
ALTER TABLE orderheader ADD CONSTRAINT FK_orderheader_promotion FOREIGN KEY
(PromotionID) REFERENCES promotion(PromotionID);
ALTER TABLE orderitem ADD CONSTRAINT FK_orderitem_orderheader FOREIGN KEY (OrderID)
REFERENCES orderheader(OrderID);
ALTER TABLE orderitem ADD CONSTRAINT FK_orderitem_title FOREIGN KEY (TitleID) REFERENCES
title>TitleID;
ALTER TABLE titleauthor ADD CONSTRAINT FK_titleauthor_title FOREIGN KEY (TitleID)
REFERENCES title>TitleID;
ALTER TABLE titleauthor ADD CONSTRAINT FK_titleauthor_author FOREIGN KEY (AuthorID)
REFERENCES author(AuthorID);
```

4. There's no correct answer here if you created the data diagram successfully. Just have fun moving around the tables to make the lines representing the relationships clearer, and be sure to hover over the lines to see how they highlight the columns represented in the relationships.

19***Creating constraints
and indexes***

We already talked about two very important constraints in chapter 18: primary key and foreign key constraints. In this chapter we will look at a few more constraints that help us ensure the integrity of the data in our tables.

We will also discuss *indexes*, which are table-related objects that help with the performance of our queries. Just as indexes in books like this one can help you quickly find the subject you are looking for, indexes in a database can help reduce the time queries need to find specific data.

I hope you've enjoyed creating tables and their associated constraints because we are about to create some more.

19.1 Constraints

By completing the examples in chapter 18, you learned that we can create constraints in two different ways: on an existing table using ALTER TABLE or when making a new table using CREATE TABLE.

There are actually two ways to create a constraint when using CREATE TABLE. We can create the constraint after all columns have been declared, as we have already done. Here is an example of the primary key constraint we created on the category table:

```
CREATE TABLE category (
    CategoryID int,
    CategoryName varchar(20),
    CONSTRAINT PK_category PRIMARY KEY (CategoryID)
);
```

We can also create a constraint as part of the declaration of the column after the data type has been declared. Here is an example of how we could have done that for the primary key on the category table:

```
CREATE TABLE category (
    CategoryID int PRIMARY KEY,
    CategoryName varchar(20)
);
```

Although this is simpler, creating a constraint in this way does not allow us to give a name for the constraint. If we create the constraint this way, the name of the constraint will be automatically generated by our RDBMS. This name will likely be some series of letters and numbers and not very indicative of our intentions with the constraint.

That being said, we don't necessarily need a name for every constraint. While it is highly recommended to create primary and foreign key constraints using the first method, other constraints we will create in this chapter are typically created using the second method.

Let's look at our first example, which is one of the most common constraints.

19.1.1 NOT NULL constraints

The NOT NULL constraint enforces that there must be data for all values in a column, which, if you think about it, would probably be the case for most columns in any given table. Tables are made to create data, and we will require many, if not all, columns in a table to have values.

As an example, imagine if we had a table in our sqlnovel database to track shipments of novels to customers. We will name the table "shipment", and it will contain the following six columns:

- **ShipmentID**, to identify each unique row in the table
- **OrderId**, to identify the order to which the shipment is related
- **ShipmentCost**, to identify the cost of the shipment in US dollars
- **ShipmentMethod**, to identify if the order was sent via parcel post (P) or express (E)
- **TrackingNumber**, to identify the tracking number provided by the shipment carrier
- **ShipmentDate**, to identify the date the shipment was sent

The data types for our columns will be:

- **ShipmentID**, int, a unique integer value
- **OrderId**, int, the same as it is in the orderheader table
- **ShipmentCost**, decimal(5,2), to accommodate numbers from 0.00 to 999.99
- **ShipmentMethod**, char(1), as this will be either a "P" or "E"
- **TrackingNumber**, varchar(20), for whatever value is provided by the shipment carrier
- **ShipmentDate**, datetime, a data value

We will also want to create a primary key constraint named "PK_shipment" on the ShipmentID column, and a foreign key constraint named "FK_shipment_orderheader" on the OrderID column that references the OrderID values in orderheader.

With all this information, we can create the shipment table using the following SQL statement:

```
CREATE TABLE shipment (
    ShipmentID int,
    OrderID int,
    ShipmentCost decimal(5,2),
    ShipmentMethod char(1),
    TrackingNumber varchar(20),
    ShipmentDate datetime,
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID) REFERENCES
orderheader(OrderID)
);
```

NOTE Don't execute this SQL yet. We're going to modify it quite a bit in this chapter.

The next thing we want to determine is which of these columns will be *nullable*, meaning they can contain null values. For every column we determine to be not nullable, we want to add a NOT NULL constraint to ensure any rows contain values for those columns. Think of nullable columns to be like the MiddleName column in the author table, where some authors will have a middle name and others will not.

Let's consider each column in the shipment table:

- **ShipmentID** will obviously not be nullable, as we can't have null values in a primary key.
- **OrderID** will also not be nullable because every shipment must relate to an order.
- **ShipmentCost** will not be nullable, as every shipment will have a cost of \$0.00 or more.
- **ShipmentMethod** will not be nullable, as we need to know how every shipment was sent.
- **TrackingNumber** will not be nullable, as each shipment will have a tracking number.
- **ShipmentDate** will not be nullable because we need to know when a shipment was sent.

After careful review, it looks like none of these columns is nullable. This means we should add a NOT NULL constraint to each column. We can do that by modifying our SQL like this to indicate which columns are NOT NULL after their data types are declared:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL,
    ShipmentMethod char(1) NOT NULL,
    TrackingNumber varchar(20) NOT NULL,
    ShipmentDate datetime NOT NULL,
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID) REFERENCES
orderheader(OrderID)
);
```

By doing this, we have now ensured every row will have a value for every column, which is what we want for this table. If someone tries to enter a row that does not have a value, then they will receive an error. This error will vary from one RDBMS to another, but in MySQL it will say that one of the columns “doesn’t have a default value.”

What does this mean? Well, *default values* are another kind of constraint. Let’s look at those next.

19.1.2 DEFAULT constraints

DEFAULT constraints allow us to use a set default value for a column if no value is specified. This default value will be used for all rows when they are created, whenever the INSERT doesn’t indicate a value for a column with a DEFAULT constraint.

The default constraint must be a *literal constant*, meaning that it will be the same literal value for every row that is inserted. This could be a number, date, or string of characters, although in MySQL and most other RDBMSs, we can also use some date and time functions as the default.

Our shipment table can benefit from this kind of constraint using one of these functions. In chapter 14, we learned about the CURRENT_DATE() function, which will return the date and time for the immediate moment. We can use this function to make sure that whenever a row is inserted into our shipment table, it will record the value for CURRENT_DATE() in the ShipmentDate column at the time the row is created.

WARNING Although fairly common, the CURRENT_DATE() function is not available in every RDBMS.
For SQL Server, use GETDATE(), for Oracle use SYSDATE, and for SQLite use date('now').

For a DEFAULT constraint, we will declare the keyword DEFAULT and then the default value, after the data types are declared for our column. Here is what our CREATE TABLE statement with this new default for ShipmentDate will look like:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL,
    ShipmentMethod char(1) NOT NULL,
    TrackingNumber varchar(20) NOT NULL,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID) REFERENCES
orderheader(OrderID)
);
```

There are two points to note about this new constraint. First, notice how we need to put parentheses around the default value of CURRENT_DATE() in our CREATE TABLE statement. The use of parentheses is not required for most RDBMSs, but it is in MySQL. This is another case where you should consult the documentation for any RDBMS you are using to make sure the syntax of your SQL statement is correct.

Also, notice that we can create more than one constraint on a column, like we are now doing with the ShipmentDate column. We don't even need to use a comma separator between our column, and in fact, we don't want to because the comma separator would indicate a new column and not a second constraint on the ShipmentDate column. This column now has both a NOT NULL and DEFAULT constraint, which is not uncommon for columns that are intended to automatically indicate the time a row was added.

NOTE Having a DEFAULT constraint on a column does not mean we don't also need the NOT NULL constraint. The DEFAULT constraint does guarantee that a value will be inserted if one is not specified, but we also need the NOT NULL constraint to ensure we do not have NULL specified as a value at the time the row is inserted.

Now let's look at another kind of constraint for a different column: the UNIQUE constraint.

19.1.3 UNIQUE constraints

UNIQUE constraints enforce that any value for a column is unique from all other values in that column. If we insert or update a value for a column with a UNIQUE constraint and that value already exists in another row, an error will occur.

UNIQUE constraints are a bit like the PRIMARY KEY constraints that we have already used, but there are a few exceptions. The main difference is that a table can contain only one PRIMARY KEY constraint, but it can contain multiple UNIQUE constraints if they are needed on columns.

Unlike PRIMARY KEY constraints, UNIQUE constraints can also include NULL values. The maximum number of NULL values a column with a UNIQUE constraint can have will depend on the RDBMS you are using, as MySQL allows multiple NULL values, while SQL Server and Oracle only allow one. If this is a concern for you, then of course, you should consult the documentation of your RDBMS, although it is rare to have a column that would require a UNIQUE constraint and still be nullable.

Since tracking numbers are UNIQUE for the shipment carrier, we want to create a UNIQUE constraint on the TrackingNumber column to ensure that a duplicate value is never used for this column. Adding this constraint is as simple as adding the word UNIQUE in the column declaration, similar to how we have done with other constraints in this chapter:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL,
    ShipmentMethod char(1) NOT NULL,
    TrackingNumber varchar(20) NOT NULL UNIQUE,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID) REFERENCES
orderheader(OrderID)
);
```

There is one final constraint we want to use in our table: the CHECK constraint.

19.1.4 CHECK constraints

CHECK constraints allow us to limit the values used in a column, by comparing them to some kind of expression. As you know by now, an expression is some combination of values, operators, or functions, so the CHECK constraint gives us quite a bit of flexibility for evaluating the validity of values for any given column.

For the shipment table, we want to add CHECK constraints to the ShipmentCost and ShipmentMethod columns. The ShipmentMethod column will require a value that is either "E" or "P", so we will write the expression to be used in our constraint as ShipmentMethod IN ('P', 'E').

WARNING The expressions we use in CHECK constraints are able to include multiple columns, so we need to always state the column (or columns) used in our expressions.

For the CHECK constraint on ShipmentCost, we want the value to be between 0.00 and 999.99, so we will write our expression as ShipmentCost BETWEEN 0.00 AND 999.99. Remember that BETWEEN is inclusive of the beginning and end values, so values of 0.00 and 999.99 are valid.

We will add our CHECK constraints similarly to how we added the DEFAULT constraint, with our constraint expression included in parentheses:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL CHECK (ShipmentCost BETWEEN 0.00 AND 999.99),
    ShipmentMethod char(1) NOT NULL CHECK (ShipmentMethod IN ('P', 'E')),
    TrackingNumber varchar(20) NOT NULL UNIQUE,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID) REFERENCES
orderheader(OrderID)
);
```

Although the CHECK constraints we are creating for the shipment table are fairly simple, as was noted earlier, these kinds of constraints can involve multiple columns. For example, we could have written a single constraint with a larger expression, to validate that the ShipmentCost was in the stated range of numeric values, and that the ShipmentMethod was one of the two acceptable values. When multiple columns are used in the expression of our constraint, we would typically want to create the constraint after all the columns, like we did with the primary and foreign key constraints.

There is one final change we want to make to our table. It is very common, and it is a bit like a DEFAULT constraint.

19.2 Automatically incrementing values for a column

We have already established that we want the ShipmentID column to be the primary key of our new shipment table. The primary key values need to be unique so that we can identify individual rows in this table.

In chapter 18, we inserted explicit values for the primary key (CategoryID) of the category table. For many tables, you will not want to use explicit values for the column determined to be used as the primary key. Instead, you will want to take advantage of a feature that every RDBMS has, which is some form of automatically incrementing value.

In MySQL, this is done using the AUTO_INCREMENT keyword. Although this is technically not a constraint, it behaves a bit like a DEFAULT constraint, in that we can insert rows into the table without specifying a value for a column set to use AUTO_INCREMENT.

The way this will work is that when we INSERT rows into the shipment table, we will omit specifying values for the ShipmentID column. The first row that is inserted this way will automatically have a value of 1 for ShipmentID, the second row inserted will have a value of 2, and so on. Having this value set to automatically populate the column with incremental values ensures the primary key will be unique and not NULL.

Additionally, we will declare AUTO_INCREMENT in our SQL the same as we have been declaring our constraints. Here is our CREATE TABLE statement, now with the ShipmentID column set to AUTO_INCREMENT:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL AUTO_INCREMENT,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL CHECK (ShipmentCost BETWEEN 0.00 AND 999.99),
    ShipmentMethod char(1) NOT NULL CHECK (ShipmentMethod IN ('P', 'E')),
    TrackingNumber varchar(20) NOT NULL UNIQUE,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID) REFERENCES
orderheader(OrderID)
);
```

Our shipment table, with all the desired constraints, is now ready to be created.

TRY IT NOW

Create the shipment table now with the preceding SQL script. We will practice using this table in the lab exercises later in this chapter.

Every constraint we've created was intended to ensure the integrity of our data, but in MySQL some of those constraints also created objects we haven't looked at yet: indexes. Indexes exist in every RDBMS, so let's look closer at these often-used objects.

19.3 Indexes

An *index* is an object that logically sorts data to make it more readable for commonly used queries, allowing queries to return data faster. There are two different kinds of indexes: clustered and non-clustered. Let's look at clustered indexes first.

19.4 Clustered Indexes

Clustered indexes are data structures that control the physical order of the rows in a table, which means this is how the data will be stored on disks or whatever storage media are used. When a table is created without any defined indexes or constraints, the rows of data are stored in no particular order, which means any query using that table will need to read every row to determine if the values contained should be retrieved, filtered, joined, etc.

As an analogy to better understand clustered indexes, think of a telephone book containing the names and telephone numbers of folks who live in your hometown. This telephone book is typically sorted by surname and then the first name of each person, with related telephone numbers and perhaps address information included with each row on any given page. If this telephone book was a table, the clustered index would be on surname and then first name, because that is how the rows in our book are organized.

We want to sort rows based on the most commonly used columns in our table, so we should be thoughtful about how we create our clustered index. Because the rows in our table can be sorted only one way, it's important to note that any table can only include one clustered index. Once created, the clustered index effectively is the table and not a separate object.

If you executed the SQL script in section 19.2, then you have already created a clustered index for the shipment table. This is because MySQL will automatically create one when we create our primary key constraint. This isn't usually a problem because the clustered index is most often created on the column or columns that make up the primary key.

As you may have noticed, when we joined tables in queries, we joined them with related primary and foreign keys. Because the RDBMS needs to read those key values to join rows in different tables, it makes sense to create clustered indexes on the primary key columns in our tables.

NOTE Even if you don't define a primary key on a table, the RDBMS will create a hidden column, often called a row identifier, that contains a unique value for every row. If you had two or more rows with identical values, the row identifier would allow the RDBMS to still know these were different rows. Because it is hidden though, users like us wouldn't typically see that hidden column, so we generally want to create primary key constraints and clustered indexes on tables requiring unique values for each row.

I said that we already had a clustered index on our table, so let's look at how we can see that a clustered index exists. In MySQL, we can see the indexes on a table in the Workbench. In the Navigator panel, we can expand our sqlnovel database, then expand the Tables, then expand our shipment table, and finally expand Indexes. As shown in figure 19.1, we have three indexes on our shipment table.

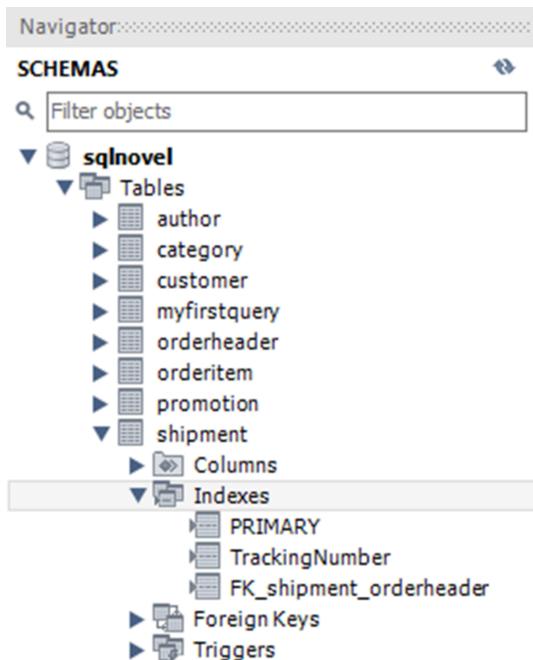


Figure 19.1 The three indexes of the shipment table in the sqlnovel database, as seen in the MySQL Workbench

We can see even more information about each index by highlighting it and viewing the Information panel, which should be below your Navigator panel. If you highlight the index named PRIMARY, then you will see the information about this index, as shown in figure 19.2

The screenshot shows the MySQL Workbench Information panel. At the top, it says "Information". Below that, it displays information for the "Index: PRIMARY". The "Definition:" section lists the following properties:

Type	BTREE
Unique	Yes
Visible	Yes
Columns	ShipmentID

Figure 19.2 Information about the PRIMARY index on the shipment table, as seen in the MySQL Workbench

Although it doesn't explicitly say "clustered index," in MySQL the index labeled PRIMARY is the clustered index. Admittedly, there isn't much information in this Information panel, but the main consideration for us is the value for Columns. This value tells us that ShipmentID is the column used for the clustered index of our table, which is the column we used to define the primary key constraint.

WARNING Because clustered indexes are not separate objects and are more like properties of a table that are often related to the primary key, every RDBMS will have different ways of handling how these are created. In some—for example, SQL Server and DB2—you can explicitly create them using an ALTER TABLE or CREATE INDEX statement, but in others, such as MySQL and PostgreSQL, you cannot. Please refer to the documentation for your specific RDBMS to see what options you have for creating clustered indexes.

Although clustered indexes are common and highly beneficial to performance, non-clustered indexes are also helpful in speeding up our queries.

19.5 Non-clustered indexes

A *non-clustered* index is different from a clustered index in two main ways. First, unlike clustered indexes, non-clustered indexes are separate objects from the tables they relate to. Second, because non-clustered indexes are separate objects, a table can contain more than one of them.

A good analogy for a non-clustered index is to think of it as a catalog system for books in a library. Non-fiction books in a library are stored using a numeric system, such as the Dewey Decimal System. However, most of us don't look for a book by its numeric value in this system, but rather by the title. We use the catalog to look up the title of our desired book, which gives us the numeric value for the book we want. We then go through the shelves of the library that are ordered by the numeric system and find our book.

Non-clustered indexes work like the catalog system in a library. They are a separate ordering of the books, in this case by title, which allows us to quickly find the title and numeric value, and then use that value to immediately go to the place in the library where the book exists. In this analogy, the numeric value would be considered as the primary key and clustered index of the non-fiction books in the library.

As you can see from our example, the catalog system (the non-clustered index) has greatly improved how quickly we can find the book we are looking for. If we didn't have the catalog, we would need to scan the entire library until we found our book. We create non-clustered indexes for the very same reason: to improve performance on finding rows of data in a table without having to read the entire table.

Although we can put all the columns we want into a non-clustered index, we will generally only have a few columns or even a single column. We typically are only searching on one or two columns, and we don't want to make our non-clustered indexes larger than necessary. The more columns they have, the more space they will take up on our storage, and the more resources they will require for any INSERT, UPDATE, and DELETE statements involving the table they relate to.

We already saw in figure 19.1 that we have two non-clustered indexes on our shipment table, so let's examine those. To look more closely at those, we can click on them in the Navigator windows and review their information in the Information panel. Let's look at the TrackingNumber index information as shown in figure 19.3.



Figure 19.3 Information about the `TrackingNumber` index on the `shipment` table, as seen in the MySQL Workbench

Because we created a UNIQUE constraint on the `TrackingNumber` column, MySQL automatically created a non-clustered index on this column. And as figure 19.3 indicates, this index has a value of "Yes" for the Unique property.

MySQL is unusual in that it automatically created the index for this column, as many other RDBMSs would not. However, because this column is required to contain unique values, we likely would have wanted to create a non-clustered index on the column anyway.

If we think about the nature of this column, containing tracking numbers for shipments, we would likely expect there to be queries looking for shipment information related to one specific `TrackingNumber`. Rather than scan the entire table every time we want to find data related to a specific `TrackingNumber`, this is exactly the kind of column we would want a non-clustered index on.

In the analogy earlier in this section, we discussed a catalog of title names in a library that worked as a sort of non-clustered index for all non-fiction books, and so here the `TrackingNumber` would serve the same purpose for the tracking numbers of our shipments.

If this index on `TrackingNumber` were not created automatically – and again, it won't be in many other RDBMSs – we could create it with the following SQL statement:

```
CREATE INDEX IX_shipment_TrackingNumber ON shipment (TrackingNumber);
```

This syntax to create a non-clustered index is common to just about every RDBMS, so we don't have to add any qualifiers about its usage.

TIP The preceding SQL statement that would create the non-clustered index on the `shipment` table uses a common but specific naming convention. That convention is a prefix of "IX_" to indicate this is an index, then an underscore and then the table name, and then another underscore and the name of the column of the index. As always, be intentional with the names of any object in your database and follow a consistent naming convention to make objects that can be easily understood by others.

Figure 19.1 indicates we have a third index in our table named FK_shipment_orderheader. This is an index that was automatically created by our foreign key constraint, which again is a behavior of MySQL that isn't necessarily the behavior of other RDBMSs. However, it is also common to create non-clustered indexes on the columns contained in foreign key constraints because these columns will be used to link tables via the relationships in the keys. Having the non-clustered indexes on the foreign key columns can reduce the need to read all the data in a table to join with other tables.

We can see the properties of this index by clicking the FK_shipment_orderheader in the Navigator panel and then viewing the Information panel as shown in figure 19.4.

The screenshot shows the MySQL Workbench Information panel. At the top, there is a tab labeled "Information". Below the tabs, the title "Index: FK_shipment_orderheader" is displayed in bold green text. Underneath the title, the heading "Definition:" is shown in bold black text. To the right of "Definition:", there is a table with the following data:

Type	BTREE
Unique	No
Visible	Yes
Columns	OrderID

Figure 19.4 Information about the FK_shipment_orderheader index on the shipment table, as seen in the MySQL Workbench

One thing to note about this index is that the value for the Unique property is "No," which is different from the other two indexes on our shipment table. Although the UNIQUE constraint we made also created an index that required unique values, it's important to note that non-clustered indexes are not required to have unique values. In the case of the FK_shipment_orderheader index, unique values will not be required because there is a possibility that we could have a one-to-many relationship between the orderheader and shipment tables, as far as OrderID values are concerned.

OK, there was a lot of database design material in this chapter. Let's briefly summarize the main points about constraints and indexes:

- Constraints are properties on one or more columns that enforce data integrity.
- NOT NULL constraints ensure no NULL values are contained in a column.
- DEFAULT constraints enter a default value if no value is specified for a column on INSERT.
- UNIQUE constraints enforce that all values in a column are different.
- CHECK constraints are used to limit the range of values that can be contained in a column.
- A clustered index defines the physical sort order of a table, typically on the primary key.
- A table can only have one clustered index.
- A non-clustered index is a separate object from the table.
- A table can have many non-clustered indexes, although every additional non-clustered index will negatively impact the performance of INSERT, UPDATE, and DELETE statements.

- In MySQL (but not every RDBMS), a clustered index is created automatically when we define a primary key constraint.
- In MySQL (but not every RDBMS), a non-clustered index is created for every UNIQUE or foreign key constraint we create.

If you're feeling up to it, try to flex your new skills with constraints and indexes in the Lab exercises.

19.6 Lab

1. Using the following values, write a SQL statement to insert rows into our new shipment table:
 - OrderID = 1001
 - ShipmentCost = 0.00
 - ShipmentMethod = 'P'
 - TrackingNumber = '1A2C3M4E'
2. Since the shipment table currently does not have a row for every order, how should we write a query to see the OrderID, OrderDate, and ShipDate for every order?
3. Could we have written the expression in section 19.1.4 differently? If so, how?
4. We want to create a report that shows the count of all orders shipped on a particular date. What constraint or index could we create to improve the performance of this report?

19.7 Lab answers

1. We do not need to specify a value for ShipmentID since it is an AUTO_INCREMENT column, and we do need to specify a value for ShipmentDate since it has a default constraint. Therefore, our SQL should look something like this:

```

INSERT shipment (
    OrderId,
    ShipmentCost,
    ShipmentMethod,
    TrackingNumber
)
VALUES (
    1001,
    0.00,
    'P',
    '1A2C3M4E'
);

```

2. Because we have values for every OrderID in orderheader but we do not have values for every OrderID in shipment, we will need to use a LEFT OUTER JOIN, like this:

```

SELECT
    oh.OrderID,
    oh.OrderDate,
    s.ShipmentDate
FROM orderheader oh
LEFT OUTER JOIN shipment s
    ON oh.OrderID = s.OrderID;

```

If we were to use an INNER JOIN instead, our result set would only include orders that have a value contained in the OrderID column in both tables.

3. There are a few different ways we could write this, including ShipmentCost >= 0.00 AND ShipmentCost <= 999.99.
4. The SQL used in our report would look something like this:

```

SELECT
    ShipmentDate,
    COUNT(ShipmentDate)
FROM shipment
WHERE ShipmentDate = @ShipmentDate
GROUP BY ShipmentDate;

```

To support this query, we would create a non-clustered index on the ShipmentDate column:

```
CREATE INDEX IX_shipment_ShipmentDate ON shipment (ShipmentDate);
```

This non-clustered index would help keep us from having to read the entire table just to determine the total for what would be a fraction of the orders shipped on any given day.

20

Reusing queries with views and stored procedures

Through 19 chapters we have written a lot of SQL queries. We've used filters, functions, aggregations, and more to find specific data. We've even added, updated, or removed data, and we've used variables to make our scripts easily able to do the same things over and over with different values.

In this chapter, we will be bringing a lot of that together by moving from just executing SQL scripts to saving our scripts as objects in the database that we or anyone else with permissions can execute. Depending on the RDBMS we are using, there are a few different objects we can use to store these scripts. For now, we're going to focus on two objects that are nearly universal: views and stored procedures.

A *view* stores a SELECT statement and provides a single result set that can be used like a table. A *stored procedure* stores one or more queries that can be executed all at once to perform nearly any required task in a database.

Let's first look at views.

20.1 Views

Views are database objects we create based on a SELECT statement. Views provide a single result set that resembles a table, which is why they are often referred to as *virtual tables*. The term *virtual tables* is also indicative of the fact that views can be used like tables in our queries.

Referring to views as virtual tables is a bit misleading, though, because views definitely are not tables and do not contain any data. It might be more helpful to think of them as "SELECT statements with a name," although that description doesn't fully describe their usefulness.

Views not only allow us to reuse a query easily, but they can reduce a complex query down to a simple and accessible object. We can then take that object and assign users permissions for whether or not they can use the view.

Rather than go on about how you should think about views, let's create one and see how we can use it.

20.1.1 Creating views

Creating any view starts with a SELECT statement. If we wanted to create a view that showed us the names titles and their category names, we could write a query like this, with the results of the query shown in figure 20.1:

```
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
```

TitleName	CategoryName
Pride and Predicates	Romance
Anne of Fact Tables	Romance
The Great GroupBy	Humor
David Emptyfield	Humor
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Red Badge of Cursors	Mystery
The Call of the While	Fantasy
A Table of Two Cities	Fantasy
The DateTime Machine	Science Fiction
The Sum Also Rises	Science Fiction
Of Mice and Metadata	Science Fiction

Figure 20.1 The results of all TitleName values from the title table and their related CategoryName values in the category table

To create a view with this query, we just need to create a SQL statement in the following order:

1. CREATE VIEW
2. Name the view
3. AS
4. Our SELECT statement

Using this easy syntax, we can create a view named `vw_TitleCategory` like this:

```
CREATE VIEW vw_TitleCategory
AS
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
```

When we create the view, it saves our `SELECT` statement to be executed whenever we want. To see the results of our view, we just need to select from the view as if it were a table:

```
SELECT *
FROM vw_TitleCategory;
```

The results of executing the preceding query would be the same as though we executed our original SQL statement, as shown in figure 20.1.

For queries that you would have to write over and over, views can help you save coding time by already having the desired result set ready to execute. Again, this view doesn't contain any data. It just calls the data via the underlying SQL Statement at the time it is executed.

But executing is not all we can do with views. There is more.

20.1.2 Filtering with views

Just as we can filter the results of a table, we could filter the results in a `WHERE` clause as we have already done many times in other queries. If we only wanted to see the titles in our view that are in the Mystery category like we see in figure 20.2, then we simply modify our `SELECT` from the view to have the appropriate filtering in a `WHERE` clause, like this:

```
SELECT *
FROM vw_TitleCategory
WHERE CategoryName = 'Mystery';
```

TitleName	CategoryName
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Red Badge of Cursors	Mystery

Figure 20.2 The results of selecting all rows from `vw_TitleCategory` with a `CategoryName` value of `Mystery`

With views, we can do nearly everything we have done with tables. We can filter results, order results, and aggregate data. We can even join views to other tables and views, but to do that with our view we will need to make some changes.

20.1.3 Joining views

As you may recall from the many times we have joined tables, we need to have relationships defined to make our joins successful. Our `vw_TitleCategory` contains two columns, but neither of them is related to any key values that form relationships with other tables. No tables in our database have any relationship that uses either the `TitleName` or `CategoryName` column as a key value.

To be able to use our view with other tables, we will need to add the key values from the underlying tables. For our view, that means adding the `TitleID` from the title table and the `CategoryID` from the category table. Now, we could add the `CategoryID` from the title table instead of from the category table, but it's a good practice to use primary keys over foreign keys in views when possible.

WARNING Just as column names in tables need to be unique, column names in views should be unique as well. If you attempt to create a view with column names that aren't unique, most RDBMSs, including MySQL, will return an error telling you there are duplicate column names.

We will modify our view using a very similar syntax to what we used to create our view, but using the `ALTER` keyword instead of `CREATE` like we did when we modified a table in chapter 18:

1. `ALTER VIEW`
2. Declare the name of the view
3. `AS`
4. Declare our modified SQL query

Using this easy syntax, we can modify our `vw_TitleCategory` to include the two additional columns with SQL that looks like this:

```
ALTER VIEW vw_TitleCategory
AS
SELECT
    t.TitleID,
    t.TitleName,
    c.CategoryID,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
```

TRY IT NOW

If you haven't created and altered the `vw_TitleCategory` yet, modify the preceding query from `ALTER VIEW` to `CREATE VIEW` to create it. We're going to use it some more in this chapter.

After executing our `ALTER VIEW` statement, we can examine the new results of our view with the following query, with the results shown in figure 20.3.

TitleID	TitleName	CategoryID	CategoryName
101	Pride and Predicates	1	Romance
104	Anne of Fact Tables	1	Romance
106	The Great GroupBy	2	Humor
109	David Emptyfield	2	Humor
102	The Join Luck Club	3	Mystery
103	Catcher in the Try	3	Mystery
110	Red Badge of Cursors	3	Mystery
107	The Call of the While	4	Fantasy
112	A Table of Two Cities	4	Fantasy
105	The DateTime Machine	5	Science Fiction
108	The Sum Also Rises	5	Science Fiction
111	Of Mice and Metadata	5	Science Fiction

Figure 20.3 The new results of selecting all rows and columns from `vw_TitleCategoryID`, which now includes the `TitleID` and `CategoryID` columns

Notice how the results of our view aren't in any particular order. This is fine, because we don't want to order the values in our view unless it is absolutely necessary. Just like a `SELECT` statement, adding an `ORDER BY` clause to a view can cause query performance to be much worse when dealing with millions of rows of data.

Not only can we now join this view to other tables and views, but we can also create calculated columns. For example, if we wanted to see how many titles were sold for each category, we can join our view in a query to the `orderitem` table using the relationship of the `TitleID` columns. Here is what that would look like in a query:

```

SELECT
    tc.CategoryName,
    SUM(oi.Quantity) AS TitlesOrdered
FROM vw_TitleCategory tc
LEFT OUTER JOIN orderitem oi
    ON tc.TitleID = oi.TitleID
GROUP BY tc.CategoryName;

```

If we execute the preceding query, we can see the results as shown in figure 20.4.

CategoryName	TitlesOrdered
Romance	27
Humor	1
Mystery	24
Fantasy	2
Science Fiction	16

Figure 20.4 The results of the number of titles ordered from each category

Views are incredibly useful, but there are some rules and caveats to their usage.

20.1.4 Considerations for views

Here are some of the larger factors you should consider as you create and use views:

- Views cannot have the same name as any other view or any other table. Keep this in mind when naming your views.
- Also, when naming views try to use a naming convention that identifies that they are views and not tables. In our example, we used the prefix of “vw_” in the name of our view. This becomes important when other users are looking at queries using views you created, because you want them to easily be able to distinguish tables from views.
- As you have seen, it’s a good idea to add columns for the primary and foreign key values to the SELECT clause of the SQL statement used by the view. Doing this allows you to relate the results of your view to other tables and views.
- In chapter 11, we examined how we can create subqueries – queries contained in parts of other queries – to find the data we want using SQL. In a similar way, views can call other views via subqueries or joins. These views used within other views are referred to as *nested views*, and they should be avoided, as they dramatically degrade query performance.
- If your view contains a calculated column, as our preceding query does, you should always create an alias for the column name. Some RDBMSs will require every column in a view to have a defined name.

- Although it may seem improbable, many RDBMSs allow for data to be updated or even inserted into views. This can be problematic, as these changes can affect data in multiple tables, so in general this is a practice that should be avoided.

Regarding that last point, views are probably not the right tool to modify any data in SQL, but stored procedures can be a wonderful tool to use for modifying data. In fact, they are a better tool for selecting data as well.

20.2 Stored procedures

Like views, stored procedures allow us to store a SQL statement in our database so we can easily reuse it over and over again. We can also assign users permissions as to whether or not they can use the stored procedure. Unlike views, stored procedures allow for even more complexity, such as executing multiple queries, passing values through variables, and more.

Let's start by turning our original SQL statement in section 20.1.1 into a stored procedure.

20.2.1 Creating stored procedures

Among nearly every RDBMS, there is a basic syntax for creating stored procedures that looks like this.

1. CREATE PROCEDURE
2. Declare the name of the stored procedure
3. Write the SQL you want the stored procedure to execute

NOTE Unfortunately, this is where the similarities in syntax end, as each RDBMS has its own subtle syntax differences for handling the creation of a stored procedure. But don't let that keep you from learning about them, because despite these differences, the usage of stored procedures is very similar across every RDBMS except SQLite, which does not support stored procedures.

To create our stored procedure in MySQL, we must first change the delimiter. When writing our first query in chapter 2, we learned that we need to add a semicolon to the end of our queries as a statement terminator to tell the RDBMS where the SQL in our query stopped. MySQL is *rigid* about the statement terminator, which can be a concern when writing stored procedures. Because they can contain multiple statements, the first semicolon encountered in our code would look like the end of the stored procedure to the RDBMS.

To work around this, we will need to briefly change the statement terminator to a value other than a semicolon. In our SQL, we will change the statement terminator to double slashes (//) by using a MySQL-specific keyword: DELIMITER, create our stored procedure containing the standard semicolon delimiter, and then use DELIMITER to change the statement terminator back to a semicolon. We will also name the procedure GetTitleCategory.

Here is what our SQL will look like to create our stored procedure to get all TitleName values and their associated CategoryName values:

```

DELIMITER //                                #A

CREATE PROCEDURE GetTitleCategory()          #B
BEGIN                                       #C
SELECT                                       #D
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
END //                                     #E

DELIMITER ;                                #F

```

#A Change the statement terminator to // using DELIMITER //. Doing this allows us to include as many SQL queries ending with semicolons as statement terminators as we need, although our new stored procedure is simple enough that it only has one query.

#B Create the procedure with CREATE PROCEDURE and the name of the stored procedure. We also included parentheses after the name, which we will discuss in the next section.

#C Indicate the start of our stored procedure with BEGIN. Using BEGIN to explicitly indicate the start of the stored procedure is not necessary in every RDBMS but it is for MySQL.

#D Write the heart of our stored procedure, which is the query to return the desired result set.

#E Indicate the end of our stored procedure with END and then the new statement terminator of //.

#F Change the statement terminator back to the semicolon with DELIMITER;.

Now that we have created our stored procedure, we can put it to use. To execute our stored procedure and see the results in figure 20.5, we will use the CALL keyword, like this:

```
CALL GetTitleCategory;
```

TitleName	CategoryName
Pride and Predicates	Romance
Anne of Fact Tables	Romance
The Great GroupBy	Humor
David Emptyfield	Humor
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Red Badge of Cursors	Mystery
The Call of the While	Fantasy
A Table of Two Cities	Fantasy
The DateTime Machine	Science Fiction
The Sum Also Rises	Science Fiction
Of Mice and Metadata	Science Fiction

Figure 20.5 The results of all TitleName values from the title table and their related CategoryName values in the category table, as returned by the stored procedure GetTitleCategory

This is a very basic stored procedure. We can do much more with stored procedures, so next we will add functionality to pass a variable and filter our results.

NOTE Not only is the writing of a stored procedure specific to an RDBMS, but so is the execution. While MySQL, PostgreSQL, and MariaDB use the CALL keyword, SQL Server and Oracle use EXEC.

20.2.2 Using variables with stored procedures

One of the biggest advantages stored procedures have over views is that stored procedures have parameters. A *parameter* is a variable that can be passed into or out of a stored procedure, and any given stored procedure can have multiple parameters that can be used for anything from filtering data to changing values in tables to determining the output results of a stored procedure.

When using parameters with a stored procedure, each parameter must have three properties defined:

- The name
- The data type
- Whether the parameter is used for input or output

The final property offers us choices on how we will use a parameter. If a parameter is defined for *input*, then a value will be passed into the stored procedure for use. If a parameter is defined for *output*, then the value of the parameter will be determined during the execution of the stored procedure and returned.

NOTE MySQL also offers a third option for parameters: INOUT. This option allows for a parameter to be passed in, modified if necessary, and then passed back out. This option is not available in every RDBMS.

We can modify our GetTitleCategory stored procedure to have an input parameter to filter on the TitleName. However, to modify a stored procedure in MySQL, we will first need to drop it, similarly to how we dropped tables.

```
DROP PROCEDURE GetTitleCategory;
```

Now we can re-create our stored procedure with an input parameter. We will name our parameter `_TitleName` so it isn't confused with the column named `TitleName`, and we will define the data type as the same data type defined for the `TitleName` column in the title table. We can see the data type of columns for any table in MySQL using `SHOW COLUMNS`. This is how we would use `SHOW COLUMNS` to find the data types of the columns in the title table, with the results shown in figure 20.6:

```
SHOW COLUMNS FROM title;
```

Field	Type	Null	Key	Default	Extra
TitleID	int	NO	PRI	NULL	
TitleName	varchar(50)	NO		NULL	
Price	decimal(5,2)	NO		NULL	
Advance	decimal(8,2)	NO		NULL	
Royalty	decimal(5,2)	YES		NULL	
PublicationDate	datetime	NO		NULL	
CategoryID	int	YES	MUL	NULL	

Figure 20.6 The data types of all columns in the title table, as returned by SHOW COLUMNS

We can see that the data type for the `TitleName` column is `varchar(50)`, so that will be the data type we will define for our input parameter. The last thing we will need to do is to make use of the parameter by using it for filtering within the stored procedure. We will add `WHERE t.TitleName = _TitleName` to accomplish this.

Our new stored procedure will be created with the following SQL:

```

DROP PROCEDURE GetTitleCategory;

DELIMITER //

CREATE PROCEDURE GetTitleCategory(
    IN _TitleName varchar(50)
)
BEGIN
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID
WHERE t.TitleName = _TitleName;
END //

DELIMITER ;

```

Now we can declare a variable and pass it to the stored procedure to return only the results for the desired TitleName value. Using the value of The Sum Also Rises, we can execute the stored procedure and view the results in figure 20.7 using the following SQL:

```

SET @TitleName = 'The Sum Also Rises';
CALL GetTitleCategory (@TitleName);

```

TitleName	CategoryName
The Sum Also Rises	Science Fiction

Figure 20.7 The results of executing GetTitleCategory using the value of The Sum Also Rises with the input parameter _TitleName

Now that we have added the input parameter _TitleName to GetTitleCategory, every execution of the stored procedure will require a value for that parameter. If we try to execute GetTitleCategory without a value for _TitleName, we will receive an error for the “Incorrect number of arguments.” In the context of a stored procedure, an *argument* is the actual value that is being passed to the parameter. GetTitleCategory is now expecting a value for every execution, and if we don’t pass one, then we are passing zero arguments. As the error correctly calculates, zero is the incorrect number when one argument is expected.

When writing a stored procedure, we might want to account for the fact that we might not have a value for an argument, so instead of not passing an argument, we can pass one that has NULL as the value. This isn’t uncommon, and as we have seen throughout this book, NULL can and will be a value that needs to be accounted for.

One way we can handle a value of NULL being passed to the `_TitleName` parameter would be to return all rows in a result set, which we can do with a simple use of the COALESCE function we first used in chapter 15. We can change the filtering in our stored procedure to `WHERE t.TitleName = COALESCE(_TitleName, t.TitleName)` to accommodate for a value of NULL being used as an argument for the `_TitleName` parameter.

With this logic, we are now able to execute our stored procedure with an argument of NULL. If a specific value is passed, our result set will still be filtered on that value, but if NULL is passed, we will now return every row where the `TitleName` equals itself, which would be every row.

Let's drop our stored procedure and re-create it with the new filtering logic that uses COALESCE. We can do all of this at once using the following SQL:

```
DROP PROCEDURE GetTitleCategory;

DELIMITER //

CREATE PROCEDURE GetTitleCategory(
IN _TitleName varchar(50)
)
BEGIN
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID
WHERE t.TitleName = COALESCE(_TitleName, t.TitleName);
END //

DELIMITER ;
```

With this new change, we can now execute our stored procedure with or without a NULL value as an argument. Using a value of NULL now returns values for all `TitleName`, as shown in figure 20.8.

```
SET @TitleName = NULL;
CALL GetTitleCategory (@TitleName);
```

TitleName	CategoryName
Pride and Predicates	Romance
Anne of Fact Tables	Romance
The Great GroupBy	Humor
David Emptyfield	Humor
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Red Badge of Cursors	Mystery
The Call of the While	Fantasy
A Table of Two Cities	Fantasy
The DateTime Machine	Science Fiction
The Sum Also Rises	Science Fiction
Of Mice and Metadata	Science Fiction

Figure 20.8 The results of all TitleName values from the title table and their related CategoryName values in the category table, as returned by the stored procedure GetTitleCategory with an argument of NULL for the _TitleName parameter

If we execute GetTitleName with an argument that isn't null, like The Sum Also Rises, we will get the filtered results for only that TitleName, as shown in figure 20.9:

```
SET @_TitleName = 'The Sum Also Rises';
CALL GetTitleCategory (@_TitleName);
```

TitleName	CategoryName
The Sum Also Rises	Science Fiction

Figure 20.9 The results of TitleName values from the title table and their related CategoryName values in the category table, as returned by the stored procedure GetTitleCategory with an argument of The Sum Also Rises for the _TitleName parameter.

TRY IT NOW

Create the final version of the GetTitleCategory stored procedure and try executing it with different values as arguments for _TitleName, including NULL.

I hope you can already see why stored procedures are a popular way to store our SQL statements in an RDBMS. Before running out and changing all our queries into stored procedures, there are a few considerations.

20.2.3 Considerations for stored procedures

As with views, there are significant factors to consider as you create and use stored procedures:

- Stored procedures can call other stored procedures, and even pass variable values back and forth. Be careful with nesting stored procedures, as this can become a headache to troubleshoot.
- As with views and other objects, have a consistent naming convention for your stored procedure so they can be easily identified as stored procedures, and so others can clearly understand the purpose of the stored procedure.
- If you are writing a stored procedure with parameters, always verify that the data types of the parameters match any data types of any columns they will be evaluated against.
- If you are using variables to pass values to the parameters of a stored procedure, always make sure the data type of your variables matches those of the stored procedure.
- Because stored procedures can contain multiple queries, use lots of clear and descriptive comments in complex stored procedures to indicate what each part of your stored procedure is intended to do.

20.3 Differences between views and stored procedures

In this chapter, we have looked at two of the most popular ways to store our SQL for reuse. As you've seen, views and stored procedures have different attributes, so let's quickly review the main differences, to help you better decide when to use either of them, as shown in table 20.1.

Table 20.1 shows some of the main differences between views and stored procedures. Use this table to help determine which one you should use.

Attributes	View	Stored procedure
Input	Does not use parameters	Can use parameters
Output	Can only return a single result set	Can return zero, one, or multiple result sets or output parameters
Multiple queries	Can only contain a single query	Can contain multiple queries
Relationships	Can be joined to other views or tables via relationships	Cannot be joined to other objects
Dependencies	Can contain a query using tables or views, but not stored procedures	Can contain queries using tables, views, or other stored procedures

Although we have covered most of what you will need to know about views, we have only scratched the surface of the capabilities of stored procedures. As you'll see in the next chapter, stored procedures can be created to read data. We can write so they can also write data and do so with logic determined by various conditions.

For now, though, let's practice what you've learned with some lab exercises.

20.4 Lab

1. Create a view named vw_Order that contains all the columns from the orderheader and orderitem tables, except the OrderID column from the orderitem table. Recall that these tables are related to each other by their OrderID columns.
2. Why do you think we should exclude the OrderID column from the orderitem table?
3. Create a stored procedure named GetOrder with the following specifications. It should:
 - a. Use the vw_Order view you just created.
 - b. Have a parameter named _OrderID and should filter the results based on matching the value of that parameter to the OrderID column of vw_Order.
 - c. Join the title table using the relationship of the TitleID columns.
 - d. Return the following columns: OrderID, OrderDate, TitleName, Quantity, and ItemPrice.
4. What value did you use for the data type of the _OrderID parameter in GetOrder, and why?
5. After creating GetOrder, what happens if you execute the following?

```
CALL GetOrder(1049)
```

20.5 Lab answers

1. The SQL for your view should look something like this:

```

CREATE VIEW vw_Order
AS
SELECT
    oh.OrderID,
    oh.CustomerID,
    oh.PromotionID,
    oh.OrderDate,
    oi.OrderItem,
    oi.TitleID,
    oi.Quantity,
    oi.ItemPrice
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID;

```

2. If we did not exclude the OrderID column from the orderitem table in our view, there would have been two columns with the name OrderID. If we then tried to create this view with two OrderID columns, the RDBMS would return an error because it wouldn't know which OrderID column to use.
3. The SQL to create the GetOrder stored procedure should look something like this:

```

DELIMITER //

CREATE PROCEDURE GetOrder(
    IN _OrderID int
)
BEGIN
SELECT
    o.OrderID,
    o.OrderDate,
    t.TitleName,
    o.Quantity,
    o.ItemPrice
FROM vw_Order o
INNER JOIN title t
    ON o.TitleID = t.TitleID
WHERE o.OrderID = _OrderID;
END //

DELIMITER ;

```

4. You should have used an integer(int) data type for the _OrderID parameter, since this is the data type of the OrderID column we will be evaluating with the parameter. If you were unsure of the data type, you could have discovered it using the following query in MySQL:

```
SHOW COLUMNS FROM orderheader;
```

5. The command should execute, returning the results shown in figure 20.10. Executing the stored procedure in this way shows we can use either variables or literal values, like 1049, when passing arguments to a parameter.

OrderID	OrderDate	TitleName	Quantity	ItemPrice
1049	2021-03-05 00:00:00	Pride and Predicates	1	6.95
1049	2021-03-05 00:00:00	The Join Luck Club	1	6.95
1049	2021-03-05 00:00:00	Catcher in the Try	1	5.95

Figure 20.10 The results of executing our new GetOrder stored procedure with an argument of the literal value 1049 passed to the _OrderID parameter

21

Making decisions in queries

Now that you know how to add, update, or remove data from a table, let's look at some of the tools SQL provides for making decisions in our queries and stored procedures.

For example, what if we want to group data and return a value of 0 if a value of the SUM is NULL? Or making the output of a query dependent on some condition? Or evaluating parameters in a stored procedure and providing conditional feedback in the output?

We're going to look at all these scenarios and more in this chapter. Let's make some decisions.

21.1 Conditional functions and expressions

Do you recall that we have already used one function in a conditional expression a few times? It was COALESCE, and we used it in chapter 15 to help us concatenate the full names of authors, and again in chapter 20 to handle NULL values for TitleName. In the first example, COALESCE allowed us to avoid a result of NULL for concatenated full names in cases where there were null values in the middle names of any authors.

21.1.1 COALESCE function

In chapter 15, we used the following query, which provided two values to COALESCE to evaluate:

```
SELECT CONCAT(FirstName, ' ', COALESCE(MiddleName, ''), ' ', LastName) as AuthorName  
FROM author;
```

The COALESCE function evaluates any number of expressions from left to right, returning the first non-null values it finds. Since the MiddleName column of the author table was the first value provided, the COALESCE function evaluated the MiddleName value for each author and then determined whether the MiddleName value was NULL. For each row where the value was not null, the MiddleName value was used in the context of the query. For each of the rows where the value was NULL, the second value – an empty string represented by a quote mark ("") – was used for concatenation.

The COALESCE function has more functionality than we used in the previous query, as we can provide it with more than two expressions to evaluate for null values. Here is a simple example using three expressions, of which the first two are NULL:

```
SELECT COALESCE(NULL, NULL, 'I am not null!') AS CoalesceTest;
```

The COALESCE function evaluated the first two expressions and determined them as null values and then returned the third expression - the string "I am not null!" - as the first non-null expression. We could have had more than three expressions evaluated by the COALESCE function, but as soon as one expression is determined to be not null, then all subsequent expressions are ignored.

TRY IT NOW

Execute the following query using the COALESCE function and see the results:

```
SELECT COALESCE(NULL, NULL, 'I am not null!', 'I am ignored!') AS CoalesceTest;
```

There is another common function that is used for evaluating nulls: IFNULL.

21.1.2 IFNULL function

The IFNULL function is nearly identical in use to the COALESCE function, with the main exception that it is limited to using only two expressions. The first expression is evaluated for null values, and if the value is determined to be NULL, then the second expression is returned. Here is an example of its usage:

```
SELECT IFNULL(NULL, 'I am not null!') AS IfNullTest;
```

NOTE The IFNULL function does not exist in all RDBMSs. In Microsoft Access or SQL Server, you will use the ISNULL function instead, and in Oracle you will use the NVL function. Although these have different names, their usage is the same as IFNULL.

Although we've been using literal values in these examples, COALESCE and IFNULL are often used with calculations that might include null values. Let's consider a common scenario where we want to see a list of all title names and determine whether they were included in any orders.

First, let's consider the tables we need to use in this query. We need the title table because it has title names, and we also need the orderitem table, as it contains the Quantity column, which represents the quantity of each item ordered in each row. We also need the orderheader table because it relates to both the title and orderitem tables with the TitleID and OrderID columns, respectively.

The way we join these tables is crucial to our output. We want to start with the title table, as we want the total quantity sold for each title, but we need to use LEFT JOINs to join the other tables since some titles may not have been included in any orders. If we used INNER JOINs to join all the tables, we would get a result set that only includes titles that have been included in orders, which isn't what we wanted with this query.

We will also want to group by TitleName from the title table and use a SUM function to get the sum of the Quantity column from orderitem for each TitleName. Our query will look something like this, and if you executed the queries in chapter 16 that added the additional titles, you should have some rows in your results that have NULL for the TotalQuantity, as shown in figure 21.1:

```
SELECT
    t.TitleName,
    SUM(oi.Quantity) AS TotalQuantity
FROM title t
LEFT JOIN orderitem oi
    ON t.TitleID = oi.TitleID
LEFT JOIN orderheader oh
    ON oh.OrderID = oi.OrderID
GROUP BY t.TitleName
ORDER BY t.TitleName;
```

TitleName	TotalQuantity
A Table of Two Cities	NULL
Anne of Fact Tables	2
Catcher in the Try	11
David Emptyfield	NULL
Of Mice and Metadata	NULL
Pride and Predicates	25
Red Badge of Cursors	NULL
The Call of the While	2
The DateTime Machine	13
The Great GroupBy	1
The Join Luck Club	13
The Sum Also Rises	3

Figure 21.1 The TitleName and the total quantity of TitleName included in orders. TitleName values that have not been included in orders will be represented by NULL.

On a sales report, NULL isn't typically what the reader would expect, so we will make a minor adjustment to our query to add IFNULL to return a value of 0 for any title that hasn't been included in an order, as shown in figure 21.2:

```
SELECT
    t.TitleName,
    ISNULL(SUM(oi.Quantity),0) AS TotalQuantity
FROM title t
LEFT JOIN orderitem oi
    ON t.TitleID = oi.TitleID
LEFT JOIN orderheader oh
    ON oh.OrderID = oi.OrderID
GROUP BY t.TitleName
ORDER BY t.TitleName;
```

TitleName	TotalQuantity
A Table of Two Cities	0
Anne of Fact Tables	2
Catcher in the Try	11
David Emptyfield	0
Of Mice and Metadata	0
Pride and Predicates	25
Red Badge of Cursors	0
The Call of the While	2
The DateTime Machine	13
The Great GroupBy	1
The Join Luck Club	13
The Sum Also Rises	3

Figure 21.2 The TitleName and the total quantity of TitleName included in orders. TitleName values that have not been included in orders will be represented by a value of 0 instead of NULL, due to our use of the IFNULL function.

Now our results will have a value of 0 instead of NULL for any title that hasn't been included in any order, which is a more useful indicator of titles included in orders.

TIP Because COALESCE has more functionality and is supported across every RDBMS, use this function for evaluating nulls instead of IFNULL or ISNULL. For this reason, we will be using COALESCE throughout this book instead of IFNULL.

COALESCE and IFNULL help evaluate expressions for null values, but what if we need to evaluate for something other than null values, or even evaluate for several different conditions? For these situations, we can use the CASE expression.

21.1.3 CASE expression

The CASE expression, often referred to in queries as a CASE statement, is a more powerful tool than COALESCE and IFNULL, since it allows us to evaluate conditions and return different values based on those conditions. It allows us to make choices on the values that are returned, using logic that emulates the English language. For example, if we wanted to find titles with prices that are \$7.95 and return a value that confirmed whether they are \$7.95, we might say something like this:

"I would like title name and price from the title table. When the price is \$7.95, then I want to say, 'This title is \$7.95.' Otherwise, I want to say, 'This title is not \$7.95.'"

We can use a CASE expression to accomplish the intention of those latter two sentences. The structure of our CASE expression has a few rules.

1. It will start with the keyword CASE.
2. It will contain one or more conditions for equality that say WHEN (some value) THEN (the desired resulting value). Because this is a test for equality, it will not evaluate values that are NULL.
3. For any value that does not meet any of the WHEN conditions, we can return a different value using ELSE. This option is not required but is often used to account for unknown or null values.
4. We conclude the CASE expression with the keyword END. If the CASE expression is used in the SELECT part of our query, we will typically want to use an alias for the column name for readability.

This may sound a little complicated, but it's very intuitive in usage. Let's look at what our SQL would look like using the preceding example, with the results shown in figure 21.3:

```
SELECT
    TitleName,
    Price,
    CASE Price
        WHEN 7.95 THEN 'This title is $7.95.'
        ELSE 'This title is not $7.95.'
    END AS IsPrice795
FROM title;
```

TitleName	Price	IsPrice795
Pride and Predicates	9.95	This title is not \$7.95.
The Join Luck Club	9.95	This title is not \$7.95.
Catcher in the Try	8.95	This title is not \$7.95.
Anne of Fact Tables	12.95	This title is not \$7.95.
The DateTime Machine	7.95	This title is \$7.95.
The Great GroupBy	10.95	This title is not \$7.95.
The Call of the While	8.95	This title is not \$7.95.
The Sum Also Rises	7.95	This title is \$7.95.
David Emptyfield	9.95	This title is not \$7.95.
Red Badge of Cursors	7.95	This title is \$7.95.
Of Mice and Metadata	8.95	This title is not \$7.95.
A Table of Two Cities	9.95	This title is not \$7.95.

Figure 21.3 The TitleName and the Price for all titles, as well as a column that is aliased as IsPrice795. The values in column IsPrice795 are the results of an evaluation of the price using a CASE expression.

As I mentioned, the evaluation in a CASE expression is for another expression, which may not necessarily be a value like those in a column. An expression could be the result of anything from concatenating two or more columns to a mathematical calculation. We can evaluate either of those or any other expression with a CASE expression.

Let's look at an example using the ROUND function we first discussed in chapter 15. If we want an expression to represent the integer values of a number such as the price of a title, we use an expression of ROUND(Price, 0) to find the nearest integer value.

We can modify our previous query to look for books with a price of around \$8.00 by using the expression ROUND(Price, 0), and then use a CASE expression to return a factual statement about the price, like this. If executed, the results will be as those shown in figure 21.4:

```

SELECT
    TitleName,
    Price,
    CASE ROUND(Price, 0)
        WHEN 8 THEN 'This title is around $8.'
        ELSE 'This title is not around $8.'
    END AS IsPriceAround8Dollars
FROM title;

```

TitleName	Price	IsPriceAround8Dollars
Pride and Predicates	9.95	This title is not around \$8.
The Join Luck Club	9.95	This title is not around \$8.
Catcher in the Try	8.95	This title is not around \$8.
Anne of Fact Tables	12.95	This title is not around \$8.
The DateTime Machine	7.95	This title is around \$8.
The Great GroupBy	10.95	This title is not around \$8.
The Call of the While	8.95	This title is not around \$8.
The Sum Also Rises	7.95	This title is around \$8.
David Emptyfield	9.95	This title is not around \$8.
Red Badge of Cursors	7.95	This title is around \$8.
Of Mice and Metadata	8.95	This title is not around \$8.
A Table of Two Cities	9.95	This title is not around \$8.

Figure 21.4 The TitleName and the Price for all titles, as well as a column that is aliased as IsPriceAround8Dollars. The values in column IsPriceAround8Dollars are the results of an evaluation of the expression ROUND(Price, 0) using a CASE expression.

The preceding two queries are examples of using simple CASE expressions, which means we are evaluating the possible values for a single expression. We can also use a searched CASE expression for more comprehensive evaluations, such as ranges of data values. With a searched CASE expression, we will evaluate one or more other expressions for whether they are true or false.

We can modify the preceding query to search for ranges of price values and see if a price is less than, equal to, or more than \$8.00 by using a searched CASE statement with the following query. The results of the query are shown in figure 21.5:

```

SELECT
    TitleName,
    Price,
    CASE
        WHEN Price < 8.00 THEN 'This title is less than $8.00.'
        WHEN Price = 8.00 THEN 'This title is $8.00.'
        WHEN Price > 8.00 THEN 'This title is more than $8.00.'
    END AS IsPriceAround8Dollars
FROM title;

```

TitleName	Price	IsPriceAround8Dollars
Pride and Predicates	9.95	This title is more than \$8.00.
The Join Luck Club	9.95	This title is more than \$8.00.
Catcher in the Try	8.95	This title is more than \$8.00.
Anne of Fact Tables	12.95	This title is more than \$8.00.
The DateTime Machine	7.95	This title is less than \$8.00.
The Great GroupBy	10.95	This title is more than \$8.00.
The Call of the While	8.95	This title is more than \$8.00.
The Sum Also Rises	7.95	This title is less than \$8.00.
David Emptyfield	9.95	This title is more than \$8.00.
Red Badge of Cursors	7.95	This title is less than \$8.00.
Of Mice and Metadata	8.95	This title is more than \$8.00.
A Table of Two Cities	9.95	This title is more than \$8.00.

Figure 21.5 The TitleName and the price for all titles, as well as a column that is aliased as IsPriceAround8Dollars. The values in column IsPriceAround8Dollars are the results of a searched CASE expression evaluating if the price value is less than, equal to, or more than \$8.00.

The expressions evaluated for being true or false are known as *Boolean expressions*. I realize we've been talking about several different kinds of expressions in this chapter, which has been complicated by CASE expressions evaluating other expressions. Please remember that these are Boolean expressions, as we will be using them again later in this chapter.

NOTE Although we have only used CASE expressions in the SELECT clause, if you need this kind of decision-making logic elsewhere in your queries, then you can use CASE expressions in other clauses in your query, including WHERE, HAVING, and ORDER BY.

Functions and expressions aren't the only tools we have in SQL for evaluation and decision-making. We also have several keywords we can use to decide whether or not we will even execute SQL statements.

21.2 Decision structures

In nearly every RDBMS there are keywords you can use to control decision-making. If you have ever used any programming language, then the good news is, those keywords should look familiar to you. And if you are new to programming, don't worry. They are very intuitive. We will first look at the one keyword necessary to start any decision-making.

21.2.1 IF and THEN

The keyword IF will be the starting point for any decision structure, which is how we refer to any SQL we write that involves deciding if we want to execute a statement. We will base our decisions on the same Boolean conditions we just used in the preceding section. This means that if a condition is true, then we want the included SQL to execute, and if that condition is false, then we do not want it to execute.

Decision structures are commonly used within stored procedures, so let's look at a simple example of how we use the IF keyword to make a decision inside a stored procedure that would add a row to the promotion table. We can write a stored procedure to add a row for a new PromotionCode, but we can create a decision structure to help avoid writing the row if a value for PromotionCode is not provided.

Before we look at the entire stored procedure, let's look at the individual parts of the SQL we are going to use inside the stored procedure to help determine if a PromotionCode value exists. Here is the first part of the stored procedure:

```
CREATE PROCEDURE AddPromotion (
    IN _PromotionID int,
    IN _PromotionCode varchar(10),
    IN _PromotionStartDate datetime,
    IN _PromotionEndDate datetime
)
BEGIN
```

Looking at this part of the stored procedure, we see that we have a name (AddPromotion) and four input parameters: _PromotionID, _PromotionCode, _PromotionStartDate, and _PromotionEndDate. The data types used for these parameters are the same as their corresponding columns in the promotion table. We also have the BEGIN keyword after we have declared the parameters to indicate where the stored procedure begins doing what we want it to do.

TIP Always create parameters with the same data types as any columns they will be used to read or write values to. Using a different data type will cause the RDBMS to do more work and will negatively affect query performance. If you don't know the data types of the columns, you can always use SHOW COLUMNS as discussed in chapter 20 to determine column data types. Although SHOW COLUMNS exists only in MySQL, there are similar keywords in every RDBMS to help you view the data types of the columns of any table.

Next, let's look at the SQL for the rest of the stored procedure:

```

IF _PromotionCode IS NOT NULL THEN
    INSERT INTO promotion (
        PromotionID,
        PromotionCode,
        PromotionStartDate,
        PromotionEndDate
    )
SELECT
    _PromotionID,
    _PromotionCode,
    _PromotionStartDate,
    _PromotionEndDate
;
END IF;
END

```

Here we have our IF keyword, which is used to say we want to determine if the condition of `_PromotionCode IS NOT NULL` is true or not. If a value other than NULL has been provided for `_PromotionCode`, then (and THEN is the other keyword we use with IF conditions) the INSERT statement will execute. If the value for `_PromotionCode` is NULL, then the subsequent INSERT statement will not execute.

We then end our conditional statement using END IF, and then we end the stored procedure with END. Let's put it all together as a single statement that we can use to create our stored procedure, and then create it so we can test the conditional logic of our decision structure. We will use the same DELIMITER commands in MySQL to change the delimiter from a semicolon so we can execute the entire stored procedure:

```

DELIMITER //

CREATE PROCEDURE AddPromotion (
    IN _PromotionID int,
    IN _PromotionCode varchar(10),
    IN _PromotionStartDate datetime,
    IN _PromotionEndDate datetime
)
BEGIN

IF _PromotionCode IS NOT NULL THEN
    INSERT INTO promotion (

```

```

PromotionID,
PromotionCode,

PromotionStartDate,

PromotionEndDate
)
SELECT
    _PromotionID,
    _PromotionCode,
    _PromotionStartDate,
    _PromotionEndDate
;
END IF;
END //


DELIMITER ;

```

If we execute this SQL, we should have our stored procedure AddPromotion ready to execute. Let's first try executing with the following arguments for the parameters:

```
CALL AddPromotion (14, '2OFF2023', '2023-01-04', '2023-02-11');
```

Executing this command should show a message of 1 row(s) affected in Output panel. We can verify the row was inserted with the following query, with the results shown in figure 21.6:

```

SELECT *
FROM promotion
WHERE PromotionID = 14;

```

PromotionID	PromotionCode	PromotionStartDate	PromotionEndDate
14	2OFF2023	2023-01-04 00:00:00	2023-02-11 00:00:00

Figure 21.6 The results of selecting any rows from the promotion table where the PromotionID is 14, which is the row we just inserted with the AddPromotion stored procedure.

Now that we verified our stored procedure works correctly when the result of our IF condition is true, let's try it if the result is false. We can do that by executing the following SQL, which will use an argument of NULL for the _PromotionCode parameter:

```
CALL AddPromotion (15, NULL, '2023-07-04', '2023-07-11');
```

Executing this command should show a message of 0 row(s) affected in Output panel. We can use a similar query to the one we used previously to confirm no row was inserted. Executing the following query will return no results:

```
SELECT *
FROM promotion
WHERE PromotionID = 15;
```

Now, we wrote the stored procedure AddPromotion, and we have examined the decision structure it uses to determine if a row should be inserted into the promotion table. But what if you weren't familiar with the internal workings of this stored procedure? Suppose you executed the previous CALL of the stored procedure that returned no results. Wouldn't you want to know why executing the stored procedure didn't work as expected?

When writing stored procedures, especially when using any kind of decision structure, we usually want the procedure to provide some sort of feedback as to the results of the decisions being made. To add functionality to our stored procedure to provide feedback or take some other action if the condition of the evaluation is false, or even if we want to add more conditions, we have some other keywords we can use.

21.2.2 ELSE

Just as IF evaluates if a condition is true, we can use ELSE to provide an alternative action to take if an evaluated condition is determined to be false or NULL. We can use ELSE in the same way that we used IF, except that it will come *after* the IF statement.

WARNING Although the use of ELSE is optional, any statement with ELSE can only follow an IF statement. If you write an ELSE statement without a preceding IF statement, you will get a syntax error.

Let's review the current decision structure we want to have in AddPromotion.

1. If _PromotionCode is not null, then we insert the values provided in the promotion table.
2. Otherwise, we don't want to insert the values, but we want to return a message explaining why the insert was skipped.

Think of ELSE as a shorter version of the word *otherwise*, meaning that after the IF condition has not been met, then this is the last thing we do.

To accomplish our goal stated in point number two, we only need to add this bit of SQL before the END IF;

```
ELSE

    SELECT 'No PromotionCode - INSERT skipped.' AS Message;
```

What this says is if a condition of false or NULL exists for the IF condition, then we will select a literal value of "No PromotionCode, INSERT skipped." that will be returned as "Message". It's about as straightforward as can be. Let's execute the following SQL to DROP AddPromotion, and then re-create with our new logic:

```
DROP PROCEDURE AddPromotion;

DELIMITER //

CREATE PROCEDURE AddPromotion (
    IN _PromotionID int,
    IN _PromotionCode varchar(10),
    IN _PromotionStartDate datetime,
    IN _PromotionEndDate datetime
)
BEGIN

IF _PromotionCode IS NOT NULL THEN
    INSERT INTO promotion (
        PromotionID,
        PromotionCode,
        PromotionStartDate,
        PromotionEndDate
    )
SELECT
    _PromotionID,
    _PromotionCode,
    _PromotionStartDate,
    _PromotionEndDate
```

```

;
ELSE
    SELECT 'No PromotionCode, INSERT skipped.' AS Message;
END IF;

END //

DELIMITER ;

```

After executing the preceding SQL, we can try executing the previous call of AddPromotion with an argument of NULL for _PromotionCode. We can now see in the results shown in figure 21.7 that we received the message included in the ELSE statement:

```
CALL AddPromotion (15, NULL, '2023-07-04', '2023-07-11');
```

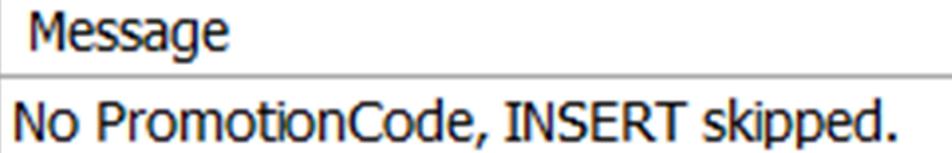


Figure 21.7 The output from the ELSE statement we added to AddPromotion, which provides a helpful message for why an insert was skipped.

So far, we have only used a simple decision structure that evaluates one particular condition, but we can also evaluate multiple conditions using IF and ELSE.

21.2.3 Multiple conditions

Our previous decision structure evaluated the value for _PromotionCode for being NOT NULL, but if we wanted AddPromotion to be a useful stored procedure then we should probably account for possible null values in the other parameters as well. To do this, we will need to change our decision structure to reflect several options for evaluation.

Let's consider this new decision structure we want to have in AddPromotion.

1. If _PromotionID is null, then we want to return a message explaining why the insert was skipped.
2. If _PromotionCode is null, then we want to return a message explaining why the insert was skipped.
3. If _PromotionStartDate is null, then we want to return a message explaining why the insert was skipped.
4. If _PromotionEndDate is null, then we want to return a message explaining why the insert was skipped.

5. Otherwise, we want to insert the values provided in the promotion table.

The SQL inside AddPromotion to handle this decision structure will need a new keyword - ELSEIF, which just means an additional IF - to handle all these additional IF statements, which would look like this:

```

IF _PromotionID IS NULL THEN
    SELECT 'No PromotionID, INSERT skipped.' AS Message;
ELSEIF _PromotionCode IS NULL THEN
    SELECT 'No PromotionCode, INSERT skipped.' AS Message;
ELSEIF _PromotionStartDate IS NULL THEN
    SELECT 'No PromotionStartDate, INSERT skipped.' AS Message;
ELSEIF _PromotionEndDate IS NULL THEN
    SELECT 'No PromotionEndDate, INSERT skipped.' AS Message;
ELSE
    INSERT INTO promotion (
        PromotionID,
        PromotionCode,
        PromotionStartDate,
        PromotionEndDate
    )
    SELECT
        _PromotionID,
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
    ;
END IF;

```

If we modify AddPromotions to include this logic using ELSEIF, then we now have a message for every possible reason for failure to insert the values.

NOTE In SQL Server, the keyword ELSEIF is represented by two words: ELSE IF.

Unfortunately, we do not have any message if the query successfully inserts the values. To add that, we will need to add another statement to our ELSE statement, and two keywords we have seen before.

Since we will now have multiple statements to execute after ELSE, we will need to group those statements into a single block with the BEGIN and END keywords. Let's look at just the ELSE statement if we want to add a message of "INSERT successful."

```
ELSE BEGIN
    INSERT INTO promotion (
        PromotionID,
        PromotionCode,
        PromotionStartDate,
        PromotionEndDate
    )
    SELECT
        _PromotionID,
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
    ;
    SELECT 'INSERT successful.' AS Message;
END;
```

Using BEGIN and END keywords allows us to group multiple statements into a single block, much like our entire stored procedure has a BEGIN and END. As you find your SQL becomes more complex, you will be using these keywords frequently – especially in stored procedures.

TRY IT NOW

DROP and CREATE the AddPromotion stored procedure using the decision structure changes discussed in section 21.2.3. After doing this, try executing it with arguments of NULL for the various parameters and verifying the correct Message is returned.

We've made a lot of decisions today, so now you should see you have a wealth of options to help you make decisions in your queries. In the next chapter, we will look at ways to use these decision-making options when evaluating individual rows of data.

21.3 Lab

1. Using CASE, write a query that will return the following three columns from the promotion table:
 - The PromotionCode column
 - The first character of the PromotionCode column with the alias "PromotionCodeLeft1"
 - The sentence "This promotion is \$X off." with the alias "PromotionDiscount" and the literal value for X replaced by the value of the second column

2. Why doesn't this query, which uses a CASE statement to attempt replacing NULL values for middle names with an empty string, work as expected?

```

SELECT
    FirstName,
    CASE MiddleName
        WHEN NULL THEN ''
        ELSE MiddleName
    END AS MiddleName,
    LastName
FROM author;
  
```

3. If you wanted to add logic to AddPromotion to skip an insert if the argument for _PromotionCode exists in the promotion table, what might that look like? (This one is a bit tricky, but try to use what you have learned to answer the question.)

21.4 Lab answers

1. You can use the LEFT function mentioned in chapter 14 to help with this. Your query might look something like this:

```

SELECT
    PromotionCode,
    LEFT(PromotionCode, 1) AS PromotionCodeLeft1,
    CASE LEFT(PromotionCode, 1)
        WHEN 1 THEN 'This promotion is for $1 off.'
        WHEN 2 THEN 'This promotion is for $2 off.'
        WHEN 3 THEN 'This promotion is for $3 off.'
    END AS PromotionDiscount
FROM promotion;
  
```

2. We cannot evaluate for equality with NULL, because NULL can never equal NULL. To work as intended, the CASE statement would need to evaluate an expression this:

```
SELECT
    FirstName,
    CASE WHEN MiddleName IS NULL THEN ''
        ELSE MiddleName
        END AS MiddleName,
    LastName
FROM author;
```

3. We might add this logic using an additional ELSEIF statement like this:

```
ELSEIF NOT EXISTS (SELECT PromotionCode FROM promotion WHERE PromotionCode =
_PromotionEndDate) THEN
    SELECT 'Duplicate PromotionCode, INSERT skipped.' AS Message;
```

This is a little more advanced than the examples we covered but can do more than check for a NULL value in a Boolean expression. As this example shows, we can even use EXISTS and NOT EXISTS to evaluate entire queries like this.

22

Using cursors

In chapter 21, we explored making decisions in queries, and we learned how we could make conditional evaluations. Using IF and THEN keywords allowed us to evaluate one or more values, and then we could decide if we wanted to do something else, like insert a row of values into a table.

In this chapter, we will look at other ways we can evaluate data and make decisions in SQL, focusing primarily on cursors. Cursors offer us not only the ability to evaluate a set of data one row or value at a time, but as you will see, they have a bit of complexity and other considerations regarding their usage.

The use of cursors in MySQL is restricted to database objects containing prepared SQL, such as stored procedures. Because of this restriction, we're going to look at some previously undiscussed features of variables and parameters before we dive into how to create and use cursors.

22.1 Reviewing variables parameters and variables

We have been using variables in every chapter since we first started using them in chapter 13, and using parameters since we began using stored procedures in chapter 20. Although they are both similar in that they are placeholders for values, they have different properties relative to their usage. Let's briefly see how we can use some of these properties.

22.1.1 Variables inside of stored procedures

In chapter 13 there was a brief mention of the way variables were declared in MySQL, and how they differed from other RDBMSs. In case you don't remember, here is the warning:

WARNING This method of declaring variables in MySQL is not universal. When using a different RDBMS such as SQL Server or PostgreSQL, you will have to first declare a user-defined variable using the DECLARE keyword and then assign it a specified data type.

Interestingly enough, in MySQL we will need to use the more common method with the DECLARE keyword noted in the warning when operating inside a stored procedure. Let's look at an example.

If we wanted to declare a variable to hold a value for TitleID from the title table outside a stored procedure, you already know we would do so like this:

```
SET @_TitleID = 101;
```

However, inside a stored procedure we would need to declare the variable and its data type using the DECLARE keyword, like this:

```
DECLARE _TitleID int;
```

We have a few options from here to assign a value such as 101 to our variable inside the stored procedure. The first option is using the SET keyword like this:

```
SET _TitleID = 101;
```

Using the SET keyword gives us some options to set this dynamically using a subquery. Here is an example of that:

```
SET _TitleID = (SELECT TitleID FROM title WHERE TitleName = 'Pride and Predicates');
```

We also have another option if we want to assign a specific value to our variable. We can use a default value, using the DEFAULT keyword:

```
DECLARE _TitleID int DEFAULT 101;
```

This final option is what we will be using in the examples of the cursors in the rest of this chapter. Now let's look at a feature of parameters we haven't previously used.

22.1.2 Output parameters

In chapter 20, we noted that parameters in stored procedures could be used for either input or output. So far, we have only used input parameters, which allow us to pass a value into a stored procedure. We have done this by declaring them with the IN keyword like this:

```
CREATE PROCEDURE GetSomeData(
    IN _TitleName varchar(50)
)
```

Declaring a parameter for output lets us take a value that has been determined inside the stored procedure and its SQL, and then pass it out to a script or even another stored procedure. This is done fairly intuitively with the OUT keyword, as in this example:

```
CREATE PROCEDURE GetSomeData(
    OUT _TitleName varchar(50)
)
```

In the examples of cursors in this chapter, we will be using output parameters, so you should have several chances to get comfortable with them and their usage.

Now that we have covered additional ways to use variables and parameters, let's talk about cursors.

22.2 Cursors

At the most basic level, a *cursor* is a database object that steps through the results of a SELECT query, allowing us to retrieve and, if desired, manipulate data one row at a time. Much like a cursor in an electronic document tells you where you are working, think of a cursor as a row pointer that enables us to loop through the result set of a query, processing individual rows for whatever desired reason.

Although cursors can be explained simply, they can be intimidating due to the complexity of their parts relative to other objects we have used, such as views and stored procedures. By that I mean we can create very simple views or stored procedures, but there aren't any simple cursors. Even the most basic cursors may look a bit intimidating when you start using them.

To make cursors more understandable, let's examine the four core components of every cursor.

22.2.1 Anatomy of a cursor

No matter how simple or complex they are, every cursor has four parts that include these descriptive keywords.

1. **DECLARE.** Just as we used DECLARE earlier in this chapter to create a variable inside a stored procedure, we will use the same keyword to create our cursor. The DECLARE part will contain the SELECT query to define the set of data our cursor will be using.
2. **OPEN.** Once the cursor is created, it must be opened to be used. Although we defined the set of data to be used by the cursor in the DECLARE part, that SELECT query doesn't actually get executed until we open the query in this part. Opening the cursor will then get the results of our SELECT query and hold them in the memory of the server while we use the cursor.

3. **FETCH.** By using the `FETCH` keyword, we will retrieve the rows from the set of data one row at a time. This is the main part of the cursor, where we populate variables, modify data, or do whatever else we intend to do with our cursor. We will work with that single row until the next time we `FETCH` another row as we loop through the result set, and we will stop fetching rows when we have reached the end of our set of data.
4. **CLOSE.** Once we have determined that we are done fetching and evaluating rows in our data set, we will `CLOSE` the cursor. Doing this releases the contents of the cursor from memory on the server.

Not only must these four parts exist for a cursor to work, but they must also be in this order.

WARNING Although it isn't required by MySQL, other RDBMSs may require you to also deallocate a cursor when you have completed using it. Please refer to the documentation of your specific RDBMS to see if this is required for any cursors you write outside of MySQL.

With these four parts of our cursor defined, let's examine how we can create and execute our first cursor.

22.2.2 Creating a cursor

Suppose we want to determine the quantity of titles that sold at the price listed in the title table, with no promotional discounts applied. To determine this, we could write a stored procedure that uses a cursor to step through each order, checking every order for any titles sold at the same price as that listed in the title table. As our cursor goes through each order, it can keep a running total of the quantity of titles sold at the list price in an output parameter, which will be returned to us with the final quantity of titles sold at the list price.

We will go through the parts of this cursor in a bit, but for now, when you take your first look at this stored procedure, just try to see if you can identify the four main parts of the cursor:

```
DELIMITER //

CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
)
BEGIN

    DECLARE _Done boolean DEFAULT FALSE;

    DECLARE _OrderID int;

    DECLARE AllOrders CURSOR FOR
```

```

SELECT OrderID
FROM orderheader;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;

SET _TotalQuantitySold = 0;

OPEN AllOrders;

GetOrders: LOOP

    FETCH AllOrders INTO _OrderID;

    SET _TotalQuantitySold = _TotalQuantitySold +
        (SELECT COALESCE(SUM(Quantity),0)
         FROM title t
         INNER JOIN orderitem oi
             ON t.TitleID = oi.TitleID
             AND t.Price = oi.ItemPrice
         WHERE oi.OrderID = _OrderID
        );

    IF _Done = TRUE THEN

        LEAVE GetOrders;
    END IF;

END LOOP GetOrders;

CLOSE AllOrders;

END //

DELIMITER ;

```

That is a lot of SQL to evaluate, so let's examine it one bit at a time. We start with changing the delimiter to two forward slashes, so we can use semicolons inside our stored procedure:

```
DELIMITER //
```

Next, we will use CREATE PROCEDURE to say we want to make a procedure named GetTitleTotalQuantitySoldListPrice. Note that our procedure has a parameter _TotalQuantitySold that not only has a data type of integer but also will be used as an output parameter. We know this because of the word OUT preceding the parameter name:

```
CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
)
BEGIN
```

We then declare two variables: one for _Done and one for OrderID. The OrderID variable will contain the OrderIDs that we will evaluate one by one. The _Done variable will be used to determine if we have completed evaluating all the rows. This variable uses a new data type called Boolean, which means it will either be true or false. We have assigned a default value of FALSE at the start of the stored procedure since we haven't finished (or even started) evaluating a data set in our cursor:

```
DECLARE _Done boolean DEFAULT FALSE;
DECLARE _OrderID int;
```

Now we have the first part of our cursor, the DECLARE part. We have declared our cursor with the name AllOrders, and our data set will include every OrderID in the orderheader table:

```
DECLARE AllOrders CURSOR FOR
SELECT OrderID
FROM orderheader;
```

We then use another DECLARE statement to tell the RDBMS that it should "handle" the condition of no more rows to evaluate ("not found set") by setting our _Done variable, which indicates we are done using the cursor, to TRUE. This will allow us to break out of the loop and stop fetching rows:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;
```

Because MySQL doesn't allow us to set a default value for parameters, we are setting the value of _TotalQuantitySold to 0 as a starting value. We will incrementally increase this value later as we find titles that were sold at the list price:

```
SET _TotalQuantitySold = 0;
```

Now we get to the second part of the cursor, where we OPEN the cursor. The query used in the DECLARE part will now be executed and the resulting data set will be stored in memory for use by the cursor:

```
OPEN AllOrders;
```

Next, we use a LOOP statement that we have named GetOrders to loop through the data set. This LOOP statement is a requirement for cursors in MySQL:

```
GetOrders: LOOP
```

NOTE Not all RDBMSs will require you to use LOOP with a cursor, so this part may be unnecessary in another RDBMS. Consult the documentation of any RDBMS you are using to better understand the various requirements for any cursor.

With the cursor opened, we now retrieve the first row of values with the FETCH part of our query. In the case of our cursor, we are only selecting the OrderID column of values, so we are fetching the first value for OrderID and assigning it to the _OrderID variable.

```
FETCH AllOrders INTO _OrderID;
```

Now that we have a value for _OrderID, we can evaluate the order to see if it includes any titles that were sold for the price listed in the title table. If it does, we will increase the value for _TotalQuantitySold by the quantity of titles that were sold for the list price. If it does not, the use of COALESCE will allow us to increment the value for _TotalQuantitySold by zero. Remember, this query is in a loop, so we will repeat it for every OrderID in the set from the DECLARE part of our cursor:

```
SET _TotalQuantitySold = _TotalQuantitySold +
    (SELECT COALESCE(SUM(Quantity),0)
     FROM title t
     INNER JOIN orderitem oi
        ON t.TitleID = oi.TitleID
        AND t.Price = oi.ItemPrice
     WHERE oi.OrderID = _OrderID
    );
```

Remember when we declared our “handler” to set the value of `_Done` to TRUE if it reached the end of the results in the cursor? Well, if that is the case here, then we are using an IF statement to exit the loop with a LEAVE keyword.

```
IF _Done = TRUE THEN
    LEAVE GetOrders;
END IF;
```

NOTE LEAVE is another keyword used in MySQL but is not used to exit cursor loops in other RDBMSs. Again, consult the documentation for any RDBMS you may be using to determine how to exit a cursor loop.

Our loop process can’t go on forever, so here we define the end of the loop. If we haven’t exited the loop via the previous statement, then we will go and fetch another value for `OrderID` at the beginning of the loop.

```
END LOOP GetOrders;
```

If we have reached this point, it means we have exited the loop, so we are done using our cursor. If we are done using our cursor, then we need to close it and release the contents from memory. We will do this with the CLOSE keyword, which brings us to the fourth and final part of our cursor:

```
CLOSE AllOrders;
```

All the work with the cursor is done now, so we need to note the end of the stored procedure with the END keyword and the non-standard statement delimiter we noted at the beginning of our script:

```
END //
```

Finally, we will change the statement delimiter back to the standard semicolon.

```
DELIMITER ;
```

If we execute all that SQL, we can now call the stored procedure with the output parameter, which we can capture in a variable named `@TotalQuantitySold`. We can then select the value of the variable to see the total quantity of titles sold at list price, with the value shown in figure 22.1.

```
CALL GetTitleTotalQuantitySoldListPrice(@TotalQuantitySold);
SELECT @TotalQuantitySold AS TotalQuantitySold;
```

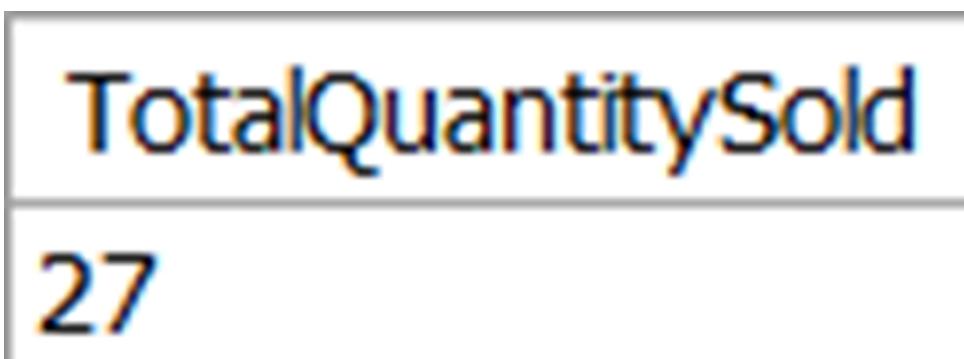


Figure 22.1 The total quantity of titles sold at the listed price, as determined by the stored procedure we wrote, which uses a cursor to determine this value.

TRY IT NOW

Create the stored procedure `GetTitleTotalQuantitySoldListPrice` and execute the preceding query to verify the total quantity of titles sold at the listed price.

Even the most basic cursors can appear to be complicated, but I hope that this walkthrough cleared up any confusion about what was happening in the individual parts of the stored procedure, especially those parts related to the cursor. If you're still a bit confused, you may be encouraged to know that you can get much of the same functionality as you would with a cursor, but in less complicated ways.

22.3 Alternatives to cursors

A common replacement for a cursor in SQL is to use something called a WHILE loop, which requires a lot less language to effectively do the same row-by-row evaluation while looping through a set of data.

22.3.1 Using WHILE

What makes the WHILE loop simpler is that we don't have to open or close our set of data. In fact, we don't even need a set of data for WHILE loop. We just need a condition for the WHILE statement that must be met to determine whether to continue in the loop.

Here is how we would rewrite our `GetTitleTotalQuantitySoldListPrice` stored procedure to use a WHILE loop instead of a cursor:

```

DROP PROCEDURE GetTitleTotalQuantitySoldListPrice;
DELIMITER //
CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
)
BEGIN
DECLARE _OrderID int;
SET _TotalQuantitySold = 0;
SET _OrderID = (SELECT MIN(OrderID) FROM orderheader);
WHILE _OrderID IS NOT NULL DO
    SET _TotalQuantitySold = _TotalQuantitySold +
        (SELECT COALESCE(SUM(Quantity),0)
        FROM title t
        INNER JOIN orderitem oi
            ON t.TitleID = oi.TitleID
            AND t.Price = oi.ItemPrice
        WHERE oi.OrderID = _OrderID
        );
    SET _OrderID = (SELECT MIN(OrderID) FROM orderheader WHERE OrderID > _OrderID);
END WHILE;
END //
DELIMITER ;

```

Let's examine the new parts so that we can understand what we are doing. First, instead of fetching the first value into our variable `_OrderID`, we have used a `SET` statement, which uses the `MIN` function to select the minimum value from the `orderheader` table – effectively the same value as the first value chosen by the previous cursor:

```
SET _OrderID = (SELECT MIN(OrderID) FROM orderheader);
```

Then we declare our `WHILE` statement, which says to keep looping through the SQL contained in the loop until `_OrderID` is `NULL`. We start the loop with a new keyword, `DO`:

```
WHILE _OrderID IS NOT NULL DO
```

NOTE Although the `DO` keyword is used in MySQL, other RDBMSs will often start with `BEGIN`. I know it may seem frustrating to keep being warned about the differences in SQL usage among RDBMSs, but this is another one of those times when you should consult the appropriate documentation to avoid syntax errors.

This next part should look familiar, as it is the identical logic we used in our cursor to incrementally add to the `_TotalQuantitySold` parameter that we used in our cursor:

```
SET _TotalQuantitySold = _TotalQuantitySold +
    (SELECT COALESCE(SUM(Quantity),0)
     FROM title t
     INNER JOIN orderitem oi
        ON t.TitleID = oi.TitleID
        AND t.Price = oi.ItemPrice
     WHERE oi.OrderID = _OrderID
    );
```

Next, we will increment the value of `_OrderID` to the next highest value using similar logic to what we used to grab the first minimum value. The difference is that now we are grabbing the minimum value that is higher than the current value, which mathematically would be the next value for `OrderID` in the `orderheader` table:

```
SET _OrderID = (SELECT MIN(OrderID) FROM orderheader WHERE OrderID > _OrderID);
```

Finally, we end the SQL included in the WHILE loop with END WHILE:

```
END WHILE;
```

This is definitely less SQL than we used for our cursor, but it is effectively doing the same thing as our cursor. However, because they both do the same thing, they could potentially create the same problem: blocking. *Blocking* is what happens when a query locks resources such as rows in a table, causing other queries that require the same resources to wait until the first query has completed its execution.

Although all the SQL we have written and executed has been on our MySQL database where we are the only users, the SQL you write outside of this book will be in a database with tens, hundreds, or even thousands of users. However, depending on database settings that you may not control, your cursor or WHILE loop in a database with more connections could cause blocking for other users, making their queries take longer or even fail if the connection has to wait too long.

TIP In many RDBMSs there will be options for cursors beyond what we have used to reduce the chances of blocking. However, the default options used by cursors will often result in blocking.

One way of working around this issue is to use *temporary tables*.

22.3.2 Temporary tables

Temporary tables are useful because they allow us to copy a set of data that might be heavily used into a separate table that exists only as long as your connection to the database exists. Once the connection is closed, the temporary tables are dropped from the database. More pertinent to cursors and WHILE loops, we can use them with no chance of blocking, as they can only be used by the queries in our connection.

The syntax for creating a temporary table in MySQL is almost identical to the syntax we used to create tables in chapter 18. The only difference is that we would add the word TEMPORARY between CREATE and TABLE.

NOTE Although temporary tables exist for nearly every RDBMS, the syntax for creating them is not universal. I hope you haven't tired of hearing this kind of message, but this is yet another case where you should consult the relevant documentation.

To avoid the chances of blocking, we could create a temporary table inside our stored procedure, populate the temporary table with the range of values we plan to use, and then direct our WHILE loop (or cursor) to loop through the temporary table.

For the previous version of GetTitleTotalQuantitySoldListPrice, here is how we could drop the existing stored procedure then then re-create it using a temporary table named orderheadertemp as previously described:

```
DROP PROCEDURE GetTitleTotalQuantitySoldListPrice;

DELIMITER //

CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
)
BEGIN
DECLARE _OrderID int;

SET _TotalQuantitySold = 0;

CREATE TEMPORARY TABLE orderheadertemp (OrderID int);

INSERT orderheadertemp (OrderID)
SELECT OrderID
FROM orderheader;

SET _OrderID = (SELECT MIN(OrderID) FROM orderheadertemp);
```

```

WHILE _OrderID IS NOT NULL DO
    SET _TotalQuantitySold = _TotalQuantitySold +
        (SELECT COALESCE(SUM(Quantity),0)
         FROM title t
         INNER JOIN orderitem oi
             ON t.TitleID = oi.TitleID
             AND t.Price = oi.ItemPrice
         WHERE oi.OrderID = _OrderID
        );
    SET _OrderID = (SELECT MIN(OrderID) FROM orderheadertemp WHERE OrderID > _OrderID);

    END WHILE;
END //
DELIMITER ;

```

Temporary tables are wonderful tools to use beyond the intention of avoiding blocking. They can be used to hold data sets that are repeatedly used in a SQL script, and they are often used to simplify complex queries by allowing us to separate them into smaller, more efficient queries. But after all this talk of cursors, WHILE loops, and temporary tables, we should probably take a moment to ask: Is all of this even necessary?

22.4 Considerations for when to use cursors

If you have read all the previous chapters and completed the exercises, then there is a good chance you have been reading this chapter with an idea of a better way to find the total quantity of titles sold at the list price. And if that is the case, you are absolutely correct. This entire request could have been handled in a much simpler way without using a cursor or a while loop:

```

SELECT COALESCE(SUM(Quantity),0) AS TotalQuantitySold
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
    AND t.Price = oi.ItemPrice;

```

Unfortunately, this is a common issue with cursors and even WHILE loops: there are usually better solutions in SQL already available to you. What makes cursors and WHILE loops inferior solutions for most query requests is that the very nature of evaluating a set of data row by row is the opposite of the way an RDBMS is designed to evaluate data, which is in sets.

22.4.1 Thinking in sets

Starting with our very first query, everything we have seen in this book, up until the latter part of chapter 21, has been what is considered set-based programming. *Set-based programming* refers to telling the RDBMS what set or sets of data you want to evaluate, and from there you let the RDBMS figure out the best way to complete the query.

We have used set-based programming with our SQL queries over and over, right up until we recently started working with IF, THEN, and ELSE keywords, along with cursors and WHILE loops. Those are considered *procedural programming*, which means they give very specific instructions to the RDBMS on what to do and how to do it. Procedural programming is actually quite common for programming languages other than SQL.

This is one reason that cursors are so long and wordy: we need to tell the RDBMS every step to take to get a cursor to evaluate data. Unfortunately, because we are telling the RDBMS what to do, using a procedural approach in SQL often results in slow performance, extensive blocking, and the consumption of more server resources than a query with a set-based approach would use.

22.4.2 Thinking about cursor usage

I don't mean to say that you should *never* use cursors, although it is possible you can solve nearly every query request without using them. But as we near the final chapters of this book, I want to encourage you to use your total knowledge of SQL and look at cursors with a bit of skepticism.

The knowledge of how to construct a cursor can certainly be useful when you have a request that has no other solution than to evaluate each row in a set individually. But those times are rare, so even when you think you need to use a cursor, take a moment to ask yourself if a set-based solution exists.

And when it comes to evaluating existing code, look at any cursor you encounter as a potential opportunity to improve performance by replacing them with set-based programming. You will likely encounter cursors frequently in existing stored procedures, as many folks with programming experience that includes procedural programming in other languages often lean on cursors in SQL instead of relying on the set-based methods you have learned throughout this book. Use your knowledge to not only reduce the complexity involved in cursors but also improve the performance of these stored procedures and reduce their resource requirements from the server.

I hope you are excited about the prospect of reviewing someone else's SQL, because that is exactly what we will be doing in the next chapter!

22.5 Lab

This lab will be a bit different from earlier labs, as we will consider a few scenarios and try to determine if we need to use a cursor to retrieve the data.

- Evaluate every TitleName in the title table to determine the quantity of each title that was ordered by customers in California. This should include customers with a value of CA for the State column in the customer table.

2. Evaluate every CustomerID in the customer table to determine if they purchased the title "Pride and Predicates". Return "Yes" if they did, and "No" if they didn't in a column named OrderedPrideAndPredicates.
3. Evaluate each order to determine if it was the first order purchased by a customer. Return the OrderID, CustomerID, and OrderDate of all orders that were the first order by any customer.
4. Evaluate every CustomerID in the customer table to see if they placed an order in the last year. If they placed an order, then execute a stored procedure named CreateThankYouMessage. This stored procedure contains a single CustomerID parameter, and it creates a message to be sent to the customer.

22.6 Lab answers

1. You do not need a cursor to determine this. You can find the requested data set from a query like this one, which uses a subquery to collect the quantity of titles ordered by customers in California:

```

SELECT
    t.TitleName,
    COALESCE(SUM(x.Quantity),0) AS QuantityFromCA
FROM title t
LEFT JOIN (
    SELECT
        oi.TitleID,
        oi.Quantity
    FROM orderitem oi
    INNER JOIN orderheader oh
        ON oi.OrderID = oh.OrderID
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.State = 'CA'
) x
    ON t.TitleID = x.TitleID
GROUP BY t.TitleName;

```

2. You don't need a cursor for this either. You can use a subquery again and a CASE statement to determine the presence of data in the subquery. You also need to be careful to use COALESCE or some other way to evaluate something other than NULL, since NULL cannot be evaluated in a CASE statement. Here we have defaulted NULL values to 0 because there is no CustomerID of 0:

```

SELECT
    c.CustomerID,
    CASE COALESCE(x.CustomerID,0)
        WHEN 0 THEN 'No'
        ELSE 'Yes'
    END AS OrderedPrideAndPredicates

FROM customer c

LEFT JOIN (
    SELECT oh.CustomerID
    FROM orderheader oh
    INNER JOIN orderitem oi
        ON oh.OrderID = oi.OrderID
    INNER JOIN title t
        ON oi.TitleID = t.TitleID
    WHERE t.TitleName = 'Pride and Predicates'
    GROUP BY oh.CustomerID
) x
ON c.CustomerID = x.CustomerID
ORDER BY c.CustomerID;

```

3. We don't need a cursor for this either. We can use another subquery to determine the first OrderID for each CustomerID and then select the desired rows from the subquery with an INNER JOIN:

```

SELECT
    oh.OrderID,
    oh.CustomerID,
    oh.OrderDate
FROM orderheader oh
INNER JOIN (
    SELECT
        ohf.CustomerID,
        MIN(ohf.OrderID) AS FirstOrderID
    FROM orderheader ohf
    GROUP BY ohf.CustomerID
) x
ON oh.OrderID = x.FirstOrderID;

```

Also, in each of these exercises, we could instead use the SQL in the subqueries to populate a temporary table and then join that temporary table instead of joining the subquery. The point is that there were choices we could use beyond the use of a cursor to achieve the desired results.

4. This is one of the few times we need to use a cursor. We will have a data set of CustomerID values, but we can't use set-based programming to complete the request since we can only provide one value at a time to the CreateThankYouMessage stored procedure.

23 *Using someone else's script*

I hope you are feeling confident with all the SQL you have learned in this book. We have covered the most basic and frequently used keywords and statements, so you should be well prepared to fulfill requests to retrieve and even manipulate data in a relational database.

As with a foreign language, though, you need to be able to listen and read as well as you speak or write. You will need to be able to read existing SQL in stored procedures and elsewhere in whatever databases your organization has, and since this book is an introduction to the SQL language, you will likely even find yourself looking for examples on the internet of SQL scripts using keywords and concepts you haven't been exposed to yet.

To practice these vital skills and apply what you have learned, we are going to review some SQL examples that were written by someone else. Know that each of the examples will work as intended, but we will have to look closely at them to even determine what the author intended. These scripts will also be going against the better practices you have learned, so we will also be considering how we can improve the SQL in the scripts.

There will be no lab at the end of the chapter, as this chapter is like one large review. There won't even be any "Try It Now" sections, although if you want, you are certainly welcome to try these scripts. The main intention here is to walk through each script, understand the intentions of the script, and improve the script based on everything you have learned in this book.

23.1 Someone else's script: creating a table

All these examples will involve a new table used to track royalty payments to authors, named `authorpayment`. The rows in the table will reflect the amount paid to each author by title and year. The presumption is that authors will be paid annually based on the sales of the titles.

23.1.1 The CREATE TABLE script

Let's start with the first script, which creates the table:

```
CREATE TABLE authorpayment (
    ID int,
    Author int,
    Title int,
    PaymentYear char(4),
    PaymentAmount decimal(7,2)
);
```

Although this isn't a particularly verbose script, I'm sure if you recall the concepts and examples we discussed in chapters 18 and 19, you can immediately spot a few things that could be corrected. Take a moment to consider the script and make some notes about what you would change.

When you are ready, keep reading and I'll share my thoughts.

23.1.2 Reviewing the CREATE TABLE script

The first thing you might notice is the column named ID, which is ambiguous. Unfortunately, naming the first column in a table as ID is somewhat common when people design database tables in a hurry and without clear intentions. We always want to be clear about a column's purpose in case this column is used in other queries, so we should rename this column AuthorPaymentID.

Also, if this AuthorPaymentID column is intended to contain unique values that form the primary key of the table, we should add a PRIMARY KEY constraint to the column, and even consider adding an AUTO_INCREMENT to automatically increment values to populate the column.

The Author column also appears to not be fully thought out. It has the same datatype as the AuthorID column used in several tables in our database, so for the sake of consistency, we should change the name to AuthorID. For the sake of data integrity, it should also contain a foreign key reference to the author table so that we only populate AuthorID in the authorpayment table with values from the author table. And since every payment has to go to an author, we want to put a NOT NULL constraint on this column.

All these same points could be said for the Title column. We should rename it to TitleID for consistency, we should create a foreign key constraint that references the TitleID values in the title table, and we should add a NOT NULL constraint to enforce a TitleID be included in every row.

The PaymentYear column is a little odd because it has a data type of char(4). This means the values will be stored as a string of characters, even though years are numeric values. We don't need to worry about any non-numeric characters occurring in years the authors will be paid, so we should use an integer (int) data type instead.

NOTE There is one case where you might want to use a character data type to store numeric values, which is if you have leading zeroes. US zip codes used for mailing addresses are a good example of this, as many zip codes start with one or more zeroes. If you stored a zip code of 03872 as an integer, it would be stored as 3872. For this reason, US zip codes are typically stored with character(char) data types.

We would also want a NOT NULL constraint on this column because every row needs to reflect a particular year, and although it's not necessary, we might want to put a CHECK constraint on the PaymentYear column to only allow values in a certain range of years. Although this would limit us, it does help to prevent many potential typos that could affect the integrity of the data. A good range for this constraint would be from the years 2000 to 2100 because, quite frankly, if this database is still being used in the year 2100 then it is well past time to upgrade.

Finally, we have the PaymentAmount column, which looks good since the data type of decimal(7,2) means we can accommodate values of payment up to \$99,999.99. We definitely want a NOT NULL constraint on this column as well, as every payment requires an amount. The only other addition we might consider here would be a CHECK constraint on the values to ensure we have only positive values in this column, as presumably, the authors will not be paid in negative amounts.

23.1.3 Improving the CREATE TABLE script

If we put all this together, our SQL to create the authorpayment table would look something like this:

```
CREATE TABLE authorpayment (
    AuthorPaymentID int NOT NULL AUTO_INCREMENT,
    AuthorID int NOT NULL,
    TitleID int NOT NULL,
    PaymentYear int NOT NULL CHECK (PaymentYear BETWEEN 2000 AND 2100),
    PaymentAmount decimal(7,2) NOT NULL CHECK (PaymentAmount BETWEEN 0.00 AND 99999.99),
    CONSTRAINT PK_AuthorPayment PRIMARY KEY (AuthorPaymentID),
    CONSTRAINT FK_authorpayment_author FOREIGN KEY (AuthorID) REFERENCES
author(AuthorID),
    CONSTRAINT FK_authorpayment_title FOREIGN KEY (TitleID) REFERENCES title>TitleID)
);
```

I hope you can see and understand how these changes will help enforce data integrity and make the table understandable and consistent with the other tables in our database. In the future, we might even consider adding a unique index to cover the AuthorID, TitleID, and PaymentYear, since it appears these values should collectively be unique for each row. We might also consider changing the primary key to use that combination of values instead of the AuthorPaymentID, which would mean we wouldn't need to create the unique index.

23.2 Someone else's script: inserting data

Next, let's look at a script that will insert rows into this new table. This stored procedure should run for each year, collecting sales information, and then determining the royalty payment to the author.

23.2.1 The INSERT stored procedure

Here is a stored procedure for you to review:

```
DELIMITER //

CREATE PROCEDURE InsertAnnualPayment(
    IN _PaymentYear int
)
BEGIN

DECLARE _Done boolean DEFAULT FALSE;
DECLARE _TitleID int;
DECLARE _AuthorID int;
DECLARE _Royalty decimal(5,2);
DECLARE _AuthorCount int;
DECLARE _TotalSales decimal(7,2);
DECLARE _PaymentAmount decimal(7,2);

DECLARE AllTitles CURSOR FOR

SELECT TitleID, AuthorID
FROM titleauthor
ORDER BY
    TitleID,
    AuthorOrder;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;

OPEN AllTitles;

GetTitles: LOOP

    FETCH AllTitles INTO _TitleID, _AuthorID;

    SET _Royalty = (
        SELECT Royalty
        FROM title
        WHERE TitleID = _TitleID
    )

```

```

);

SET _AuthorCount = (
    SELECT COUNT(AuthorID)
    FROM titleauthor
    WHERE TitleID = _TitleID
);

SET _TotalSales = (
    SELECT SUM(orderitem.Quantity * orderitem.ItemPrice)
    FROM orderheader
    INNER JOIN orderitem
        ON orderheader.OrderID = orderitem.OrderID
    WHERE orderitem.TitleID = _TitleID
        AND YEAR(orderheader.OrderDate) = _PaymentYear
);

SET _PaymentAmount = COALESCE(CONVERT(((_TotalSales * (_Royalty/100))/_AuthorCount),
decimal(7,2)), 0.00);

IF _PaymentAmount > 0.00 THEN
    INSERT authorpayment (
        AuthorID,
        TitleID,
        PaymentYear,
        PaymentAmount
    )
    SELECT
        _AuthorID,
        _TitleID,
        _PaymentYear,
        _PaymentAmount;
END IF;

IF _Done = TRUE THEN

    LEAVE GetTitles;
END IF;

END LOOP GetTitles;

```

```
CLOSE AllTitles;

END //

DELIMITER ;
```

Take a moment to review the stored procedure and maybe even take some notes before proceeding.

23.2.2 Reviewing the INSERT stored procedure

As you might guess from what was said in chapter 21, the first thing to notice about this stored procedure is that it uses a cursor to determine the annual royalty payments. I hope when you see this cursor that you start considering if there is any way to turn this row-by-row evaluation into a set-based evaluation instead.

To determine if we can replace the cursor, we should first look at what the stored procedure is doing with the cursor. Let's go through each section of the stored procedure.

The start of the stored procedure looks fairly standard. It looks as though an integer value for the input parameter `_PaymentYear` is required, but that is the only parameter:

```
DELIMITER //

CREATE PROCEDURE InsertAnnualPayment(
    IN _PaymentYear int
)
BEGIN
```

After that, a lot of variables are declared. Although they all seem to have sensible data types, on closer inspection, we can see the `_Royalty` value does not match the decimal (5,2) data type used for the `Royalty` column in the title table. Having mismatched data types can lead to data errors or inconsistencies. Also, as we go through the cursor you will see that many if not all of these variables may not be necessary:

```
DECLARE _Done boolean DEFAULT FALSE;
DECLARE _TitleID int;
DECLARE _AuthorID int;
DECLARE _Royalty int;
DECLARE _AuthorCount int;
DECLARE _TotalSales decimal(7,2);
DECLARE _PaymentAmount decimal(7,2);
```

The cursor with the name AllTitles is declared. Notice that unlike the cursors we used in chapter 22, this cursor uses two columns instead of one:

```
DECLARE AllTitles CURSOR FOR

SELECT TitleID, AuthorID
FROM titleauthor
ORDER BY
    TitleID,
    AuthorOrder;
```

The handler variable `_Done` is declared to determine when to exit the cursor loop:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;
```

We then OPEN the cursor, where the results of the query used by the cursor are retrieved:

```
OPEN AllTitles;
```

Next, we create the LOOP used by the cursor, named GetTitles:

```
GetTitles: LOOP
```

The first results of the cursor are fetched, and the values are assigned to the variables `_TitleID` and `_AuthorID`:

```
FETCH AllTitles INTO _TitleID, _AuthorID;
```

Then we start assigning values to other variables. The first assignment is the value for `_Royalty`, which is assigned for the particular `_TitleID` value. We probably don't need to have a separate query for this value, as we're going to use it later in the stored procedure to calculate the value for the `_PaymentAmount` variable. We could just as easily use the value in the title table instead of populating and using this variable:

```
SET _Royalty =
    SELECT Royalty
    FROM title
    WHERE TitleID = _TitleID
    );
```

The value for `_AuthorCount` is determined for the particular `_TitleID`. Having this determined by a separate query might not be a bad idea, although if we use a GROUP BY on the `titleauthor` table and group by `TitleID`, we might be able to use an INNER JOIN to include the `Royalty` amount noted above as well. Because `Royalty` is a column in the `title` table, we know there will be a one-to-one relationship with the results grouped by `TitleID` in the `titleauthor` table.

```
SET _AuthorCount = (
    SELECT COUNT(AuthorID)
    FROM titleauthor
    WHERE TitleID = _TitleID
);
```

The value for `_TotalSales`, which represents the sales in terms of dollars, is determined by taking the SUM of the `Quantity` multiplied by the `Price` for the title for all orders placed in the `_PaymentYear`. The rows that match the `_PaymentYear` are calculated by using the `YEAR` function to determine the year of every value of the `OrderDate` column in the `orderheader` table. This also makes sense to have as a separate query, but we probably should avoid using the `YEAR` function in this way. Although it appears convenient, as noted in chapter 14, functions can be very inefficient if there are ever millions of rows to evaluate in these tables.

```
SET _TotalSales = (
    SELECT SUM(orderitem.Quantity * orderitem.ItemPrice)
    FROM orderheader
    INNER JOIN orderitem
        ON orderheader.OrderID = orderitem.OrderID
    WHERE orderitem.TitleID = _TitleID
        AND YEAR(orderheader.OrderDate) = _PaymentYear
);
```

In the last variable assignment, the calculation to determine the value for `_PaymentAmount`, which is the amount to be paid to the author based on their royalty, is calculated using the `_TotalSales`, `_Royalty`, and `_AuthorCount` values. We're getting to the heart of the cursor used by the stored procedure, and it seems as though other than the values for `_TotalSales` and `AuthorCount`, we have a lot of unnecessary queries being used to determine values because the author wasn't thinking about a set-based solution.

```
SET _PaymentAmount = COALESCE(CONVERT((( _TotalSales * (_Royalty/100))/_AuthorCount),
decimal(7,2)), 0.00);
```

If the value for `_PaymentAmount` is greater than 0, then a row representing the payment is inserted into the `authorpayment` table. Although necessary within a cursor, if we used a set-based approach, we wouldn't need this IF...THEN... because the results from properly used INNER JOINs would exclude any titles and authors that did not have a titles sold, and therefore, would result in no payment:

```
IF _PaymentAmount > 0.00 THEN
    INSERT authorpayment (
        AuthorID,
        TitleID,
        PaymentYear,
        PaymentAmount
    )
SELECT
    _AuthorID,
    _TitleID,
    _PaymentYear,
    _PaymentAmount;
END IF;
```

If we have fetched all the values for the rows in our cursor, then here is where we will exit the loop:

```
IF _Done = TRUE THEN

    LEAVE GetTitles;
END IF;
```

At the end of the stored procedure, we end the LOOP, close the cursor, use END to represent the end of all actions in the stored procedure, and then change the delimiter back to the standard semicolon:

```
END LOOP GetTitles;
CLOSE AllTitles;
END //
DELIMITER ;
```

After reviewing the entire stored procedure, we should be able to make improvements that not only eliminate the use of a cursor and make the RDBMS do less work but also eliminate the need for any variables.

23.2.3 Improving the INSERT stored procedure

The first thing we want to do is rewrite the sections that have queries we want to keep. In our review, it was noted that we could not only use a query to determine the count of authors for each title, which will be used to calculate the royalty payment, but we could also include the Royalty values from the title table as well. The query, which will be used in a subquery, could look like this:

```
SELECT
    t.TitleID,
    t.Royalty,
    COUNT(ta.AuthorID) AS AuthorCount
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
GROUP BY

    t.TitleID,
    t.Royalty
```

It was also noted that we could use the logic to determine the sales of a title per year in dollars in a subquery and that we didn't want to use a function on the OrderDate column of the orderheader table, which would cause extra work for the RDBMS. We can use some different date functions to take the value for year and then calculate starting and ending dates for the value of the _PaymentYear parameter.

The first is MAKEDATE, which allows us to make a date of the first day of the year using only a value for year. We will make the date for the first day of the chosen year like this:

```
MAKEDATE(_PaymentYear, 1)
```

We could use MAKEDATE to select the last day of the year by replacing the value of 1 with 365, but that only works for most years. Leap years have 366 days, and since we don't know if the value passed for _PaymentYear is a leap year, it would be better to just use the end of our date range as anything less than the start of the next year. To calculate that, we will use the DATE_ADD function like this:

```
DATE_ADD(MAKEDATE(@Year, 1), INTERVAL 1 YEAR)
```

NOTE Although these functions don't exist in every RDBMS, they all have similar functions with different names that can help you determine a date from parts like the year, as well as those that can help you make calculations with dates.

With the range of dates that will replace the use of the YEAR function sorted, here is what this query to determine the total sales per title, which will also be used in a subquery, could look like:

```
SELECT
    oi.TitleID,
    SUM(oi.Quantity * oi.ItemPrice) AS TotalSales
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
WHERE oh.OrderDate >= MAKEDATE(@Year, 1)
    AND oh.OrderDate < DATE_ADD(MAKEDATE(@Year, 1), INTERVAL 1 YEAR)
GROUP BY
    oi.TitleID
```

We just need one more query that calculates the payment amount. Having this value for each AuthorID and TitleID for the chosen year will allow us to populate the author payment table. And since the two previous queries that will be used for subqueries both include TitleID, these should be simple to join.

Using the logic to calculate the payment amount from the original stored procedure, we can put all the logic together in our stored procedure with one query. Because there is a bit of complexity involved, let's add a few comments to our stored procedure to explain our intentions:

```
DELIMITER //

CREATE PROCEDURE InsertAnnualPayment(
    IN _PaymentYear int
)
BEGIN

    INSERT authorpayment (
        AuthorID,
        TitleID,
        PaymentYear,
        PaymentAmount
    )
    /* Calculate the total royalty per author */
    SELECT
        ta.AuthorID,
        ta.TitleID,
        _PaymentYear,
        CONVERT(((sales.TotalSales * (royalty.Royalty/100))/royalty.AuthorCount),
        decimal(7,2)) AS RoyaltyPerAuthor
    FROM titleauthor ta
```

```

INNER JOIN (
    /* Determine annual sales by title */

    SELECT
        oi.TitleID,
        SUM(oi.Quantity * oi.ItemPrice) AS TotalSales
    FROM orderheader oh
    INNER JOIN orderitem oi
        ON oh.OrderID = oi.OrderID
    WHERE oh.OrderDate >= MAKEDATE(@Year, 1)
        AND oh.OrderDate < DATE_ADD(MAKEDATE(@Year, 1), INTERVAL 1 YEAR)
    GROUP BY
        oi.TitleID
    ) sales
    ON ta.TitleID = sales.TitleID
INNER JOIN (
    /* Determine the royalty and count of authors */
    SELECT
        t.TitleID,
        t.Royalty,
        COUNT(ta2.AuthorID) AS AuthorCount
    FROM title t
    INNER JOIN titleauthor ta2
        ON t.TitleID = ta2.TitleID
    GROUP BY

        t.TitleID,
        t.Royalty
    ) royalty
    ON ta.TitleID = royalty.TitleID;
END //
DELIMITER ;

```

Now we have a stored procedure with no cursor, no variables, and no queries beyond what we need. We can populate the authorpayment table using set-based programming, which should be the goal whenever writing SQL. We don't have to worry about mismatched data types, and we aren't using functions in ways that would affect performance.

The stored procedure is greatly improved. But could we improve this even more?

23.2.4 Improving the INSERT stored procedure even more

You have learned a lot through 23 chapters, so don't be afraid to consider other keywords and techniques when making further improvements to someone else's script. There are actually a few more parts of this stored procedure that we might be able to improve. First is the use of the subqueries.

Although the two subqueries in this new version of the stored procedure perform better than the cursor, this might not be the best solution if our orderheader and orderitem tables contain millions of records. It might be a good idea to replace the subquery that calculates the TotalSales for each title in a given year with a temporary table. Although this means writing more data to a temporary table, the subsequent INNER JOIN may read less data than all that data read in the subquery. Although this isn't required for the current set of data that has just a few rows in each table, you will encounter many situations where, with a bit of testing, you will find temporary tables can help performance.

Another possible improvement would be related to the `_PaymentYear` parameter. The single `_PaymentYear` parameter used by the stored procedure doesn't offer much flexibility, so we might consider replacing it with two parameters, for start date and end date of a range of dates. This would allow for annual, quarterly, monthly, or even a custom range of values. Consider making any script more flexible if you can in anticipation of more diverse query requests in the future.

And finally, depending on the usage of this data, it might be worth replacing the `InsertAnnualPayment` stored procedure and the `authorpayment` table with a view that calculates the author and title royalty for every sale. Remember that a view is simply a stored query to be called upon whenever we want, and if we remove the consideration for a specific date range, then we could build a query to calculate the royalty of every `orderitem` with a query like this:

```

SELECT
    ta.AuthorID,
    ta.TitleID,
    oh.OrderID,
    oh.OrderDate,
    CONVERT(((SUM(oi.Quantity * oi.ItemPrice) * (t.Royalty/100))/ac.AuthorCount),
decimal(7,2)) AS RoyaltyPerAuthor
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
INNER JOIN orderitem oi
    ON ta.TitleID = oi.TitleID
INNER JOIN orderheader oh
    ON oi.OrderID = oh.OrderID
INNER JOIN (
    /* Determine the royalty and count of authors */
    SELECT
        TitleID,
        COUNT(AuthorID) AS AuthorCount
    FROM titleauthor

    GROUP BY
        TitleID
    ) ac
    ON ta.TitleID = ac.TitleID
GROUP BY
    ta.AuthorID,
    ta.TitleID,
    oh.OrderID
    oh.OrderDate;

```

Having this view would eliminate the redundancy of all the data in the authorpayment table. It would also allow us to query the view for whatever values of AuthorID, TitleID, and OrderID we want, as well as query any range of OrderDate values. This doesn't allow us to persist the data the way the authorpayment table does, but if we don't need it persisted, then a view would be a great option.

This is the key idea you should take away from this chapter: you have options. Throughout this month of lunches, you have learned dozens of keywords and concepts, and how to use them effectively. I hope you found reviewing these scripts helpful, and that this reinforced a higher level of confidence in your abilities to use the SQL language.

24 Never the end

We have arrived at the final chapter of *Learn SQL in a Month of Lunches*. I hope this book has been useful to you and has convinced you that even with little or no programming experience, anyone can learn to write useful SQL queries.

Starting with chapter 1, the goal has always been for you to be immediately effective at writing SQL queries. With all the concepts and keywords we have discussed and used, you should feel confident enough to write queries that satisfy a wide range of requests. You now know of different ways to filter, join, and group data, as well as how to modify data and even create objects such as tables and stored procedures. With all that we have covered, I'm confident you have learned enough to understand most examples of SQL that someone else has written.

Still, the end of this book is hardly the end of your exploration of the SQL language. In fact, this is truly just the beginning because the more you work with SQL, the more you will discover new and exciting keywords and objects you can use. So where to go next? Well, here are a few ideas.

24.1 More SQL

As you surely must have noticed in the MySQL Workbench Navigator, there is a section named "Functions" that we never addressed. Although we worked with dozens of functions throughout this book, such as CONCAT and COALSECE, none of those functions is included there. This is because those are system functions, and the "Functions" section is for user-defined functions. That's right, you can create your own functions! As you progress in your experience with SQL, you will definitely encounter requests that require evaluating values or expressions with a function that doesn't yet exist.

Another consideration for future learning is window functions. Although they aren't available in every RDBMS, when working with an RDBMS that includes them, you can perform powerful calculations such as running totals, ranking, and percentile for each row. In some ways they operate like a cursor, but without the locking, blocking, and excessive resource utilization.

And when your SQL needs to be constructed based on unknown conditions, many RDBMSs will offer you the option to use dynamic SQL. As strange as this sounds, *dynamic SQL* allows you to create a string of SQL that can later be executed. Although somewhat unusual, this technique allows for another level of flexibility with your SQL, which can be useful when your query needs to dynamically change filtering clauses or even the names of tables being queried.

These are just a few of the many tools and techniques still to be discovered on your SQL journey. So where should you go next to learn about these and your other possible steps?

24.2 Other SQL resources

As with any language, whether spoken to a computer or to another person, the best way to increase your skill level is through practice. The reason I have added labs in nearly every chapter of this book is to get you to practice thinking about SQL and using it to solve problems. You could continue to practice using the sqlnovel database to write SQL that does things like insert new rows in the orderheader and orderitem tables or retrieve data for sales by category. The possibilities for practice are limited only by your imagination.

Then again, your immediate need may be to work with an RDBMS other than MySQL, in which case you can install a tool that allows you to work with that RDBMS and find a sample database to use for practicing SQL queries. There are free sample databases available for any RDBMS, so use your favorite search engine to find them and write your own practice queries. The more you practice writing SQL, the easier it will be to respond effectively to any request.

If you found this book helpful, take a look at the RDBMS-specific books from Manning that can help you improve your SQL skills, such as *100 SQL Server Mistakes and How to Avoid Them* and *PostgreSQL Mistakes and How to Avoid Them*. As has been noted throughout this book, every RDBMS is slightly different in SQL syntax, and an RDBMS-specific book can help increase your depth of knowledge in ways that can be beneficial to your career. While it's good to have a broad knowledge of the SQL language in the ways we have discussed throughout this book, obtaining most of your experience in a particular RDBMS could allow you to showcase yourself as an expert in that particular flavor of SQL.

Perhaps you will discover that you want to move beyond writing queries that retrieve data and learn about creating databases. If so, consider books such as *Understanding Databases* and *Grokking Relational Database Design* that cover ways to effectively design databases that perform well and scale with the massive amounts of data modern databases need to contain. The design of databases is something we didn't discuss in-depth, but understanding the capabilities and limitations of a database is also an important skill in today's world.

Above all, no matter what you choose next, be curious and don't stop learning.

24.3 Farewell

It's been my pleasure to help you begin what I hope is a long-lasting journey using the SQL language. Whatever the future holds for you, I congratulate you now on all the work you've done so far, and I wish you the best for whatever comes next!