O'REILLY®

Third Edition

Learning SQL

Generate, Manipulate, and Retrieve Data

Early Release
RAW & UNEDITED

Alan Beaulieu

Learning SQL

THIRD EDITION

Generate, Manipulate, and Retrieve Data

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Alan Beaulieu

Learning SQL

by Alan Beaulieu

Printed in the United States of America.

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

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

Acquisitions Editor: Jessica Haberman

Development Editor: Jeff Bleiel

Production Editor: Deborah Baker

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

May 2020: Third Edition

Revision History for the Early Release

- 2019-12-11: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492057611 for release details.

978-1-492-05754-3

[LSI]

Preface

Programming languages come and go constantly, and very few languages in use today have roots going back more than a decade or so. Some examples are Cobol, which is still used quite heavily in mainframe environments, and C, which is still quite popular for operating system and server development and for embedded systems. In the database arena, we have SQL, whose roots go all the way back to the 1970s.

SQL is the language for generating, manipulating, and retrieving data from a relational database. One of the reasons for the popularity of relational databases is that properly designed relational databases can handle huge amounts of data. When working with large data sets, SQL is akin to one of those snazzy digital cameras with the high-power zoom lens in that you can use SQL to look at large sets of data, or you can zoom in on individual rows (or anywhere in between). Other database management systems tend to break down under heavy loads because their focus is too narrow (the zoom lens is stuck on maximum), which is why attempts to dethrone relational databases and SQL have largely failed. Therefore, even though SQL is an old language, it is going to be around for a lot longer and has a bright future in store.

Why Learn SQL?

If you are going to work with a relational database, whether you are writing applications, performing administrative tasks, or generating reports, you will need to know how to interact with the data in your database. Even if you are using a tool that generates SQL for you, such as a reporting tool, there may be times when you need to bypass the automatic generation feature and write your own SQL statements.

Learning SQL has the added benefit of forcing you to confront and understand the data structures used to store information about your organization. As you become comfortable with the tables in your database, you may find yourself proposing modifications or additions to your database schema.

Why Use This Book to Do It?

The SQL language is broken into several categories. Statements used to create database objects (tables, indexes, constraints, etc.) are collectively known as SQL *schema statements*. The statements used to create, manipulate, and retrieve the data stored in a database are known as the SQL *data statements*. If you are an administrator, you will be using both SQL schema and SQL data statements. If you are a programmer or report writer, you may only need to use (or be *allowed* to use) SQL data statements. While this book demonstrates many of the SQL schema statements, the main focus of this book is on programming features.

With only a handful of commands, the SQL data statements look deceptively simple. In my opinion, many of the available SQL books help to foster this notion by only skimming the surface of what is possible with the language. However, if you are going to work with SQL, it behooves you to understand fully the capabilities of the language and how different features can be combined to produce powerful results. I feel that this is the only book that provides detailed coverage of the SQL language without the added benefit of doubling as a “door stop” (you know, those 1,250-page “complete references” that tend to gather dust on people’s cubicle shelves).

While the examples in this book run on MySQL, Oracle Database, and SQL Server, I had to pick one of those products to host my sample database and to format the result sets returned by the example queries. Of the three, I chose MySQL because it is freely obtainable, easy to install, and simple to administer. For those readers using a different server, I ask that you download and install MySQL and load the sample database so that you can run the examples and experiment with the data.

Structure of This Book

This book is divided into 15 chapters and 3 appendixes:

> Chapter 1, explores the history of computerized databases, including the rise of the relational model and the SQL language.

> Chapter 2, demonstrates how to create a MySQL database, create the tables used for the examples in this book, and populate the tables with data.

> Chapter 3, introduces the `select`statement and further demonstrates the most common clauses (`select`, `from`, `where`).

> Chapter 4, demonstrates the different types of conditions that can be used in the `where`clause of a `select`, `update`, or `delete`statement.

> Chapter 5, shows how queries can utilize multiple tables via table joins.

> Chapter 6, is all about data sets and how they can interact within queries.

Chapter 7, demonstrates several built-in functions used for manipulating or converting data.

Chapter 8, shows how data can be aggregated.

Chapter 9, introduces the subquery (a personal favorite) and shows how and where they can be utilized.

Chapter 10, further explores the various types of table joins.

Chapter 11, explores how conditional logic (i.e., if-then-else) can be utilized in `select`, `insert`, `update`, and `delete`statements.

Chapter 12, introduces transactions and shows how to use them.

Chapter 13, explores indexes and constraints.

Chapter 14, shows how to build an interface to shield users from data complexities.

Chapter 15, demonstrates the utility of the data dictionary.

Appendix A shows the database schema used for all examples in the book.

Appendix B demonstrates some of the interesting non-ANSI features of MySQL's SQL implementation.

Appendix C shows solutions to the chapter exercises.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Used for filenames, directory names, and URLs. Also used for emphasis and to indicate the first use of a technical term.

`Constant width`

Used for code examples and to indicate SQL keywords within text.

`Constant width italic`

Used to indicate user-defined terms.

plainUPPERCASE

Used to indicate SQL keywords within example code.

`Constant width bold`

Indicates user input in examples showing an interaction. Also indicates emphasized code elements to which you should pay particular attention.

NOTE

Indicates a tip, suggestion, or general note. For example, I use notes to point you to useful new features in Oracle9*i*.

WARNING

Indicates a warning or caution. For example, I'll tell you if a certain SQL clause might have unintended consequences if not used carefully.

Using Code Examples

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

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning SQL*, Third Edition, by Alan Beaulieu. Copyright 2020 Alan Beaulieu, 978-1-492-05761-1."

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

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

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

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/9781492057611*.

Emails us with comments or technical questions at *bookquestions@oreilly.com*.

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

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Chapter 1. A Little Background

Before we roll up our sleeves and get to work, it would be helpful to survey the history of database technology in order to better understand how relational databases and the SQL language evolved. Therefore, I'd like to start by introducing some basic database concepts and looking at the history of computerized data storage and retrieval.

> **NOTE**
>
> For those readers anxious to start writing queries, feel free to skip ahead to Chapter 3, but I recommend returning later to the first two chapters in order to better understand the history and utility of the SQL language.

Introduction to Databases

A *database* is nothing more than a set of related information. A telephone book, for example, is a database of the names, phone numbers, and addresses of all people living in a particular region. While a telephone book is certainly a ubiquitous and frequently used database, it suffers from the following:

- Finding a person's telephone number can be time-consuming, especially if the telephone book contains a large number of entries.

- A telephone book is indexed only by last/first names, so finding the names of the people living at a particular address, while possible in theory, is not a practical use for this database.
- From the moment the telephone book is printed, the information becomes less and less accurate as people move into or out of a region, change their telephone numbers, or move to another location within the same region.

The same drawbacks attributed to telephone books can also apply to any manual data storage system, such as patient records stored in a filing cabinet. Because of the cumbersome nature of paper databases, some of the first computer applications developed were *database systems*, which are computerized data storage and retrieval mechanisms. Because a database system stores data electronically rather than on paper, a database system is able to retrieve data more quickly, index data in multiple ways, and deliver up-to-the-minute information to its user community.

Early database systems managed data stored on magnetic tapes. Because there were generally far more tapes than tape readers, technicians were tasked with loading and unloading tapes as specific data was requested. Because the computers of that era had very little memory, multiple requests for the same data generally required the data to be read from the tape multiple times. While these database systems were a significant improvement over paper databases, they are a far cry from what is possible with today's technology. (Modern database systems can manage petabytes of data, accessed by clusters of servers each caching tens of gigabytes of that data in high-speed memory, but I’m getting a bit ahead of myself.)

Nonrelational Database Systems

> **NOTE**
>
> This section contains some background information about pre-relational database systems. For those readers eager to dive into SQL, feel free to skip ahead a couple of pages to the next section.

Over the first several decades of computerized database systems, data was stored and represented to users in various ways. In a *hierarchical database system*, for example, data is represented as one or more tree structures. Figure 1-1 shows how data relating to George Blake’s and Sue Smith’s bank accounts might be represented via tree structures.

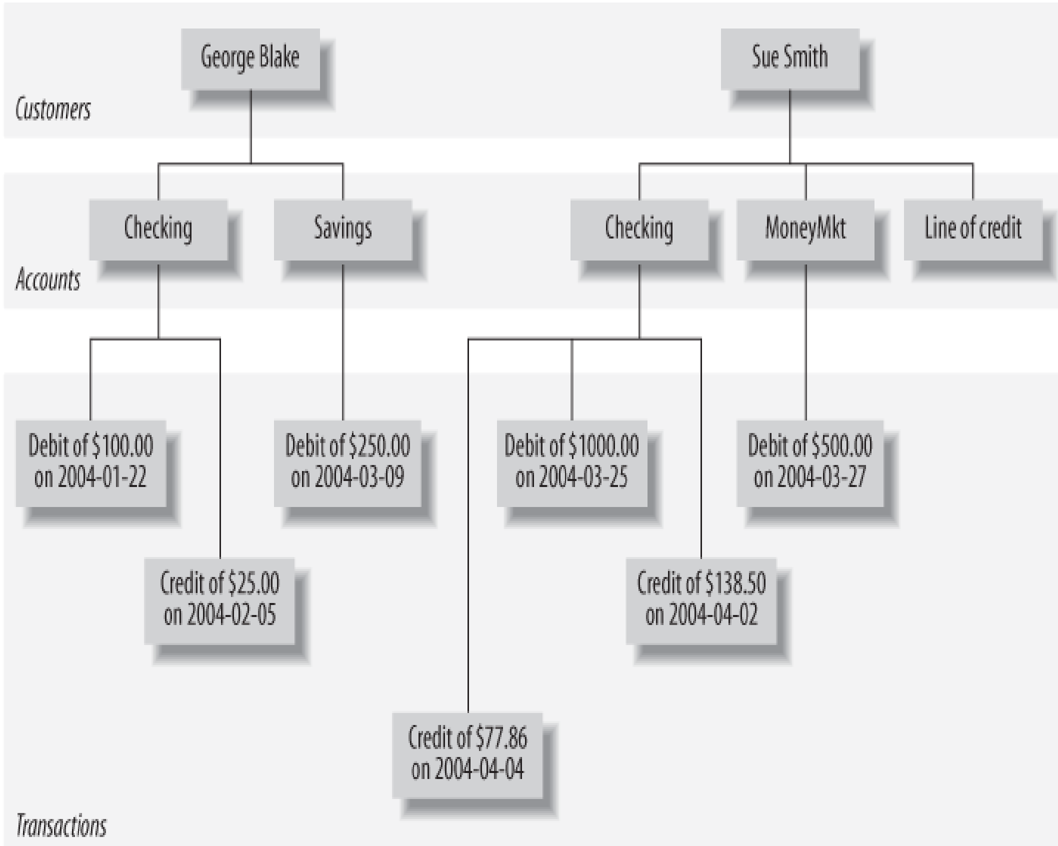

Figure 1-1. Hierarchical view of account data

George and Sue each have their own tree containing their accounts and the transactions on those accounts. The hierarchical database system provides tools for locating a particular customer's tree and then traversing the tree to find the desired accounts and/or transactions. Each node in the tree may have either zero or one parent and zero, one, or many children. This configuration is known as a *single-parent hierarchy*.

Another common approach, called the *network database system*, exposes sets of records and sets of links that define relationships between different records. Figure 1-2 shows how George's and Sue's same accounts might look in such a system.

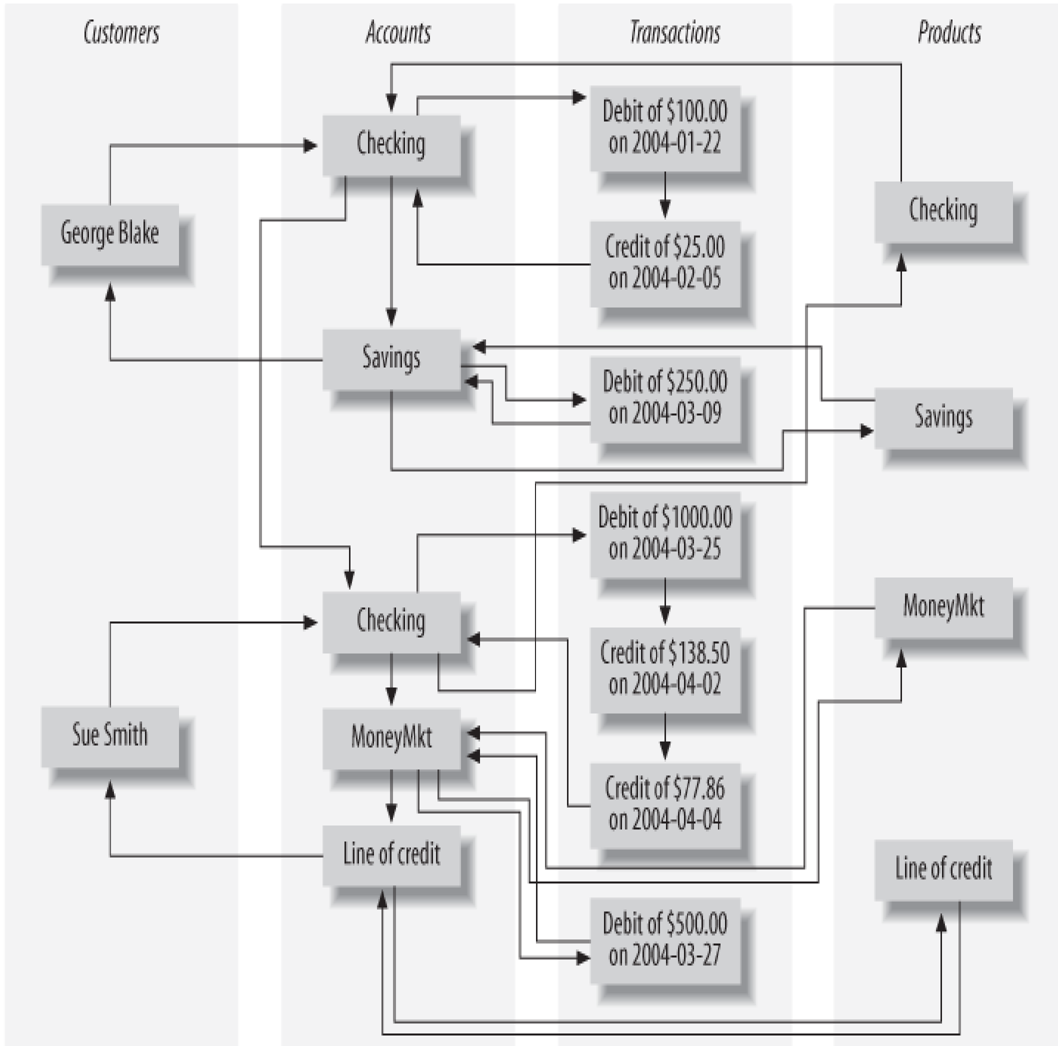

Figure 1-2. Network view of account data

In order to find the transactions posted to Sue's money market account, you would need to perform the following steps:

1. Find the customer record for Sue Smith.
2. Follow the link from Sue Smith's customer record to her list of accounts.
3. Traverse the chain of accounts until you find the money market account.

4. Follow the link from the money market record to its list of transactions.

One interesting feature of network database systems is demonstrated by the set of `product` records on the far right of Figure 1-2. Notice that each `product` record (Checking, Savings, etc.) points to a list of `account` records that are of that product type. `Account` records, therefore, can be accessed from multiple places (both `customer` records and `product` records), allowing a network database to act as a *multiparent hierarchy*.

Both hierarchical and network database systems are alive and well today, although generally in the mainframe world. Additionally, hierarchical database systems have enjoyed a rebirth in the directory services realm, such as Microsoft's Active Directory and the open-source Apache Directory Server. Beginning in the 1970s, however, a new way of representing data began to take root, one that was more rigorous yet easy to understand and implement.

The Relational Model

In 1970, Dr. E. F. Codd of IBM's research laboratory published a paper titled "A Relational Model of Data for Large Shared Data Banks" that proposed that data be represented as sets of *tables*. Rather than using pointers to navigate between related entities, redundant data is used to link records in different tables. Figure 1-3 shows how George's and Sue's account information would appear in this context.

Customer

cust_id	fname	lname
1	George	Blake
2	Sue	Smith

Account

account_id	product_cd	cust_id	balance
103	CHK	1	$75.00
104	SAV	1	$250.00
105	CHK	2	$783.64
106	MM	2	$500.00
107	LOC	2	0

Product

product_cd	name
CHK	Checking
SAV	Savings
MM	Money market
LOC	Line of credit

Transaction

txn_id	txn_type_cd	account_id	amount	date
978	DBT	103	$100.00	2004-01-22
979	CDT	103	$25.00	2004-02-05
980	DBT	104	$250.00	2004-03-09
981	DBT	105	$1000.00	2004-03-25
982	CDT	105	$138.50	2004-04-02
983	CDT	105	$77.86	2004-04-04
984	DBT	106	$500.00	2004-03-27

Figure 1-3. Relational view of account data

There are four tables in Figure 1-3 representing the four entities discussed so far: `customer`, `product`, `account`, and `transaction`. Looking across the top of the `customer` table in Figure 1-3, you can see three *columns*: `cust_id` (which contains the customer's ID number), `fname` (which contains the customer's first name), and `lname` (which contains the customer's last name). Looking down the side of the `customer` table, you can see two *rows*, one containing George Blake’s data and the other containing Sue Smith’s data. The number of columns that a table may contain differs from server to server, but it is generally large enough not to be an issue (Microsoft SQL Server, for example, allows up to 1,024 columns per table). The number of rows that a table may contain is more a matter of physical limits (i.e., how much disk drive space is available) and maintainability (i.e., how large a table can get before it becomes difficult to work with) than of database server limitations.

Each table in a relational database includes information that uniquely identifies a row in that table (known as the *primary key*), along with additional information needed to describe the entity completely. Looking again at the `customer` table, the `cust_id` column holds a different number for each customer; George Blake, for example, can be uniquely identified by customer ID #1. No other customer will ever be assigned that identifier, and no other information is needed to locate George Blake’s data in the `customer` table.

> **NOTE**
>
> Every database server provides a mechanism for generating unique sets of numbers to use as primary key values, so you won’t need to worry about keeping track of what numbers have been assigned.

While I might have chosen to use the combination of the `fname` and `lname` columns as the primary key (a primary key consisting of two or more columns is known as a *compound key*), there could easily be two or more people with the same first and last names that have accounts at the bank. Therefore, I chose to include the `cust_id` column in the `customer` table specifically for use as a primary key column.

> **NOTE**
>
> In this example, choosing `fname`/`lname` as the primary key would be referred to as a *natural key*, whereas the choice of `cust_id` would be referred to as a *surrogate key*. The decision whether to employ natural or surrogate keys is up to the database designer, but in this particular case the choice is clear, since a person's last name may change (such as when a person adopts a spouse's last name), and primary key columns should never be allowed to change once a value has been assigned.

Some of the tables also include information used to navigate to another table; this is where the "redundant data" mentioned earlier comes in. For example, the `account` table includes a column called `cust_id`, which contains the unique identifier of the customer who opened the account, along with a column called `product_cd`, which contains the unique identifier of the product to which the account will conform. These columns are known as *foreign keys*, and they serve the same purpose as the lines that connect the entities in the hierarchical and network versions of the account information. If you are looking at a particular account record and want to know more information about the customer who opened the account, you would take the value of the `cust_id` column and use it to find the appropriate row in the `customer` table (this process is known, in relational database lingo, as a *join*; joins are introduced in Chapter 3 and probed deeply in Chapters Chapter 5 and Chapter 10).

It might seem wasteful to store the same data many times, but the relational model is quite clear on what redundant data may be stored. For example, it is proper for the `account` table to include a column for the unique identifier of the customer who opened the account, but it is not proper to include the customer's first and last names in the `account` table as well. If a customer were to change her name, for example, you want to make sure that there is only one place in the database that holds the customer's name; otherwise, the data might be changed in one place but not another, causing the data in the database to be unreliable. The proper place for this data is the `customer` table, and only the `cust_id` values should be included in other tables. It is also not proper for a single column to contain multiple pieces of information, such as a `name` column that contains both a person's first and last names, or an `address` column that contains street, city, state, and zip code information. The process of refining a database design to ensure that each independent piece of information is in only one place (except for foreign keys) is known as *normalization*.

Getting back to the four tables in Figure 1-3, you may wonder how you would use these tables to find George Blake's transactions against his checking account. First, you would find George Blake's unique identifier in the `customer` table. Then, you would find the row in the `account` table whose `cust_id` column contains George's unique identifier and whose `product_cd` column matches the row in the `product` table whose `name` column equals "Checking." Finally, you would locate the rows in the `transaction` table whose `account_id` column matches the unique identifier from the `account` table. This might sound complicated, but you can do it in a single command, using the SQL language, as you will see shortly.

Some Terminology

I introduced some new terminology in the previous sections, so maybe it's time for some formal definitions. Table 1-1 shows the terms we use for the remainder of the book along with their definitions.

Table 1-1. Terms and definitions

Term	Definition
Entity	Something of interest to the database user community. Examples include customers, parts, geographic locations, etc.
Column	An individual piece of data stored in a table.
Row	A set of columns that together completely describe an entity or some action on an entity. Also called a record.
Table	A set of rows, held either in memory (nonpersistent) or on permanent storage (persistent).
Result set	Another name for a nonpersistent table, generally the result of an SQL query.
Primary key	One or more columns that can be used as a unique identifier for each row in a table.

Term	Definition
Foreign key	One or more columns that can be used together to identify a single row in another table.

What Is SQL?

Along with Codd's definition of the relational model, he proposed a language called DSL/Alpha for manipulating the data in relational tables. Shortly after Codd's paper was released, IBM commissioned a group to build a prototype based on Codd's ideas. This group created a simplified version of DSL/Alpha that they called SQUARE. Refinements to SQUARE led to a language called SEQUEL, which was, finally, shortened to SQL.While SQL began as a language used to manipulate data in relational databases, it has evolved (as you will see toward the end of this book) to be a language for manipulating data across various database technologies.

SQL is now over 40 years old, and it has undergone a great deal of change along the way. In the mid-1980s, the American National Standards Institute (ANSI) began working on the first standard for the SQL language, which was published in 1986. Subsequent refinements led to new releases of the SQL standard in 1989, 1992, 1999, 2003, 2006, 2008, 2011, 2016. Along with refinements to the core language, new features have been added to the SQL language to incorporate object-oriented functionality, among other things. The later standards focus on the integration of related technologies, such as XML and JSON.

SQL goes hand in hand with the relational model because the result of an SQL query is a table (also called, in this context, a *result set*). Thus, a new permanent table can be created in a relational database simply by storing the result set of a query. Similarly, a query can use both permanent tables and the result sets from other queries as inputs (we explore this in detail in Chapter 9).

One final note: SQL is not an acronym for anything (although many people will insist it stands for "Structured Query Language"). When referring to the language, it is equally acceptable to say the letters individually (i.e., S. Q. L.) or to use the word *sequel*.

SQL Statement Classes

The SQL language is divided into several distinct parts: the parts that we explore in this book include *SQL schema statements*, which are used to define the data structures stored in the database; *SQL data statements*, which are used to manipulate the data structures previously defined using SQL schema statements; and *SQL transaction statements*, which are used to begin, end, and roll back transactions (covered in Chapter 12). For example, to create a new table in your database, you would use the SQL schema statement `create table`, whereas the process of populating your new table with data would require the SQL data statement `insert`.

To give you a taste of what these statements look like, here’s an SQL schema statement that creates a table called `corporation`:

```
CREATE TABLE corporation
 (corp_id SMALLINT,
  name VARCHAR(30),
  CONSTRAINT pk_corporation PRIMARY KEY (corp_id)
 );
```

This statement creates a table with two columns, `corp_id` and `name`, with the `corp_id` column identified as the primary key for the table. We probe the finer details of this statement, such as the different data types available with MySQL, in Chapter 2. Next, here’s an SQL data statement that inserts a row into the `corporation` table for Acme Paper Corporation:

```
INSERT INTO corporation (corp_id, name)
VALUES (27, 'Acme Paper Corporation');
```

This statement adds a row to the `corporation` table with a value of `27` for the `corp_id` column and a value of `Acme Paper Corporation` for the `name` column.

Finally, here’s a simple `select` statement to retrieve the data that was just created:

```
mysql< SELECT name
    -> FROM corporation
    -> WHERE corp_id = 27;
+------------------------+
| name                   |
+------------------------+
| Acme Paper Corporation |
+------------------------+
```

All database elements created via SQL schema statements are stored in a special set of tables called the *data dictionary*. This “data about the database” is known collectively as *metadata* and is explored in Chapter 15. Just like tables that you create yourself, data dictionary tables can be queried via a `select` statement, thereby allowing you to discover the current data structures deployed in the database at runtime. For example, if you are asked to write a report showing the new accounts created last month, you could either hardcode the names of the columns in the `account` table that were known to you when you wrote the report, or query the data dictionary to determine the current set of columns and dynamically generate the report each time it is executed.

Most of this book is concerned with the data portion of the SQL language, which consists of the `select`, `update`, `insert`, and `delete` commands. SQL schema statements is demonstrated in Chapter 2, where the sample database used throughout this book is generated. In general, SQL schema statements do not require much discussion apart from their syntax, whereas SQL data statements, while few in number, offer numerous opportunities for detailed study. Therefore, while I try to introduce you to many of the SQL schema statements, most chapters in this book concentrate on the SQL data statements.

SQL: A Nonprocedural Language

If you have worked with programming languages in the past, you are used to defining variables and data structures, using conditional logic (i.e., if-then-else) and looping constructs (i.e., do while … end), and breaking your code into small, reusable pieces (i.e., objects, functions, procedures). Your code is handed to a compiler, and the executable that results does exactly (well, not always *exactly*) what you programmed it to do. Whether you work with Java, Python, Scala, or some other *procedural* language, you are in complete control of what the program does.

> **NOTE**
>
> A procedural language defines both the desired results and the mechanism, or process, by which the results are generated. Nonprocedural languages also define the desired results, but the process by which the results are generated is left to an external agent.

With SQL, however, you will need to give up some of the control you are used to, because SQL statements define the necessary inputs and outputs, but the manner in which a statement is executed is left to a component of your database engine known as the *optimizer*. The optimizer's job is to look at your SQL statements and, taking into account how your tables are configured and what indexes are available, decide the most efficient execution path (well, not always the *most* efficient). Most database engines will allow you to influence the optimizer's decisions by specifying *optimizer hints*, such as suggesting that a particular index be used; most SQL users, however, will never get to this level of sophistication and will leave such tweaking to their database administrator or performance expert.

With SQL, therefore, you will not be able to write complete applications. Unless you are writing a simple script to manipulate certain data, you will need to integrate SQL with your favorite programming language. Some database vendors have done this for you, such as Oracle's PL/SQL language, MySQL's stored procedure language, and Microsoft's Transact-SQL language. With these languages, the SQL data statements are part of the language's grammar, allowing you to seamlessly integrate database queries with procedural commands. If you are using a non-database-specific language such as Java or Python, however, you will need to use a toolkit/API to execute SQL statements from your code. Some of these toolkits are provided by your database vendor, whereas others are created by third-party vendors or by open-source providers. Table 1-2 shows some of the available options for integrating SQL into a specific language.

Table 1-2. SQL integration toolkits

Language	Toolkit
Java	JDBC (Java Database Connectivity)
C#	ADO.NET (Microsoft)
Ruby	Ruby DBI
Python	Python DB
Go	Package database/sql

If you only need to execute SQL commands interactively, every database vendor provides at least a simple command-line tool for submitting SQL commands to the database engine and inspecting the results. Most vendors provide a graphical tool as well that includes one window showing your SQL commands and another window showing the results from your SQL commands. Additionally, there are 3rd-party tools such as SQuirrel, which will connect via a JDBC connection to many different database servers. Since the examples in this book are executed against a MySQL database, I use the `mysql` command-line tool that is included as part of the MySQL installation to run the examples and format the results.

SQL Examples

Earlier in this chapter, I promised to show you an SQL statement that would return all the transactions against George Blake's checking account. Without further ado, here it is:

```
SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM individual i
  INNER JOIN account a ON i.cust_id = a.cust_id
  INNER JOIN product p ON p.product_cd = a.product_cd
  INNER JOIN transaction t ON t.account_id = a.account_id
WHERE i.fname = 'George' AND i.lname = 'Blake'
  AND p.name = 'checking account';

+--------+-------------+---------------------+--------+
| txn_id | txn_type_cd | txn_date            | amount |
+--------+-------------+---------------------+--------+
|     11 | DBT         | 2008-01-05 00:00:00 | 100.00 |
+--------+-------------+---------------------+--------+
1 row in set (0.00 sec)
```

Without going into too much detail at this point, this query identifies the row in the `individual` table for George Blake and the row in the `product` table for the "checking" product, finds the row in the `account` table for this individual/product combination, and returns four columns from the `transaction` table for all transactions posted to this account. If you happen to know that George Blake's customer ID is 8 and that checking accounts are designated by the code `'CHK'`, then you can simply find George Blake's checking account in the `account` table based on the customer ID and use the account ID to find the appropriate transactions:

```
SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM account a
  INNER JOIN transaction t ON t.account_id = a.account_id
WHERE a.cust_id = 8 AND a.product_cd = 'CHK';
```

I cover all of the concepts in these queries (plus a lot more) in the following chapters, but I wanted to at least show what they would look like.

The previous queries contain three different *clauses*: `select`, `from`, and `where`. Almost every query that you encounter will include at least these three clauses, although there are several more that can be used for more specialized purposes. The role of each of these three clauses is demonstrated by the following:

```
SELECT /* one or more things */ ...
FROM /* one or more places */ ...
WHERE /* one or more conditions apply */ ...
```

NOTE

Most SQL implementations treat any text between the `/*` and `*/` tags as comments.

When constructing your query, your first task is generally to determine which table or tables will be needed and then add them to your `from` clause. Next, you will need to add conditions to your `where` clause to filter out the data from these tables that you aren't interested in. Finally, you will decide which columns from the different tables need to be retrieved and add them to your `select` clause. Here's a simple example that shows how you would find all customers with the last name "Smith":

```
SELECT cust_id, fname
FROM individual
WHERE lname = 'Smith';
```

This query searches the `individual` table for all rows whose `lname` column matches the string `'Smith'` and returns the `cust_id` and `fname` columns from those rows.

Along with querying your database, you will most likely be involved with populating and modifying the data in your database. Here's a simple example of how you would insert a new row into the `product` table:

```
INSERT INTO product (product_cd, name)
VALUES ('CD', 'Certificate of Depysit')
```

Whoops, looks like you misspelled "Deposit." No problem. You can clean that up with an `update` statement:

```
UPDATE product
SET name = 'Certificate of Deposit'
WHERE product_cd = 'CD';
```

Notice that the `update` statement also contains a `where` clause, just like the `select` statement. This is because an `update` statement must identify the rows to be modified; in this case, you are specifying that only those rows whose `product_cd` column matches the string

`'CD'` should be modified. Since the `product_cd` column is the primary key for the `product` table, you should expect your `update` statement to modify exactly one row (or zero, if the value doesn't exist in the table). Whenever you execute an SQL data statement, you will receive feedback from the database engine as to how many rows were affected by your statement. If you are using an interactive tool such as the `mysql` command-line tool mentioned earlier, then you will receive feedback concerning how many rows were either:

- Returned by your `select` statement
- Created by your `insert` statement
- Modified by your `update` statement
- Removed by your `delete` statement

If you are using a procedural language with one of the toolkits mentioned earlier, the toolkit will include a call to ask for this information after your SQL data statement has executed. In general, it's a good idea to check this info to make sure your statement didn't do something unexpected (like when you forget to put a `where` clause on your `delete` statement and delete every row in the table!).

What Is MySQL?

Relational databases have been available commercially for over two decades. Some of the most mature and popular commercial products include:

- Oracle Database from Oracle Corporation
- SQL Server from Microsoft
- DB2 Universal Database from IBM

All these database servers do approximately the same thing, although some are better equipped to run very large or very-high-throughput databases. Others are better at handling objects or very large files or XML documents, and so on. Additionally, all these servers do a pretty good job of complying with the latest ANSI SQL standard. This is a good thing, and I make it a point to show you how to write SQL statements that will run on any of these platforms with little or no modification.

Along with the commercial database servers, there has been quite a bit of activity in the open source community in the past two decades with the goal of creating a viable alternative to the commercial database servers. Two of the most commonly used open source database servers are PostgreSQL and MySQL. The MySQL server is available for free, and I have found it to be extremely simple to download and install. For these reasons, I have decided that all examples for this book be run against a MySQL (version 8.0) database, and that the `mysql` command-line tool be used to format query results. Even if you are already using another server and never plan to use MySQL, I urge you to install the latest MySQL server, load the sample schema and data, and experiment with the data and examples in this book.

However, keep in mind the following caveat:

This is not a book about MySQL's SQL implementation.

Rather, this book is designed to teach you how to craft SQL statements that will run on MySQL with no modifications, and will run on recent releases of Oracle Database, DB2, and SQL Server with few or no modifications.

SQL Unplugged

A great deal has happened in the database world during the decade between the 2nd and 3rd editions of this book. While relational databases are still heavily used and will continue to be for some time, new database technologies have emerged to meet the needs of companies like Amazon and Google. These technologies include Hadoop, Spark, NoSQL, and NewSQL, which are distributed, scalable systems typically deployed on clusters of commodity servers. While it is beyond the scope of this book to explore these technologies in detail, they do all share something in common with relational databases: SQL.

Since organizations frequently store data using multiple technologies, there is a need to unplug SQL from a particular database server and provide a service which can span multiple databases. For example, a report may need to bring together data stored in Oracle, Hadoop, JSON files, CSV files, and Unix log files. A new generation of tools have been built to meet this type of challenge, and one of the most promising is Apache Drill, which is an open-source query engine which allows users to write queries which can access data stored in most any database or file system. We will explore Apache Drill in Chapter 18, SQL and Big Data.

What’s in Store

The overall goal of the next four chapters is to introduce the SQL data statements, with a special emphasis on the three main clauses of the `select` statement. Additionally, you will see many examples that use the sakila schema (introduced in the next chapter), which will be used for all examples in the book. It is my hope that familiarity with a single database will allow you to get to the crux of an example without your having to stop and examine the tables being used each time. If it becomes a bit tedious working with the same set of tables, feel free to augment the sample database with additional tables, or invent your own database with which to experiment.

After you have a solid grasp on the basics, the remaining chapters will drill deep into additional concepts, most of which are independent of each other. Thus, if you find yourself getting confused, you can always move ahead and come back later to revisit a chapter. When you have finished the book and worked through all of the examples, you will be well on your way to becoming a seasoned SQL practitioner.

For readers interested in learning more about relational databases, the history of computerized database systems, or the SQL language than was covered in this short introduction, here are a few resources worth checking out:

- C.J. Date’s *Database in Depth: Relational Theory for Practitioners* (O’Reilly)
- C.J. Date’s *An Introduction to Database Systems*, Eighth Edition (Addison-Wesley)
- C.J. Date’s *The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E. F. Codd’s Contribution to the Field of Database Technology* (Addison-Wesley)
- *http://en.wikipedia.org/wiki/Database_management_system*

Chapter 2. Creating and Populating a Database

This chapter provides you with the information you need to create your first database and to create the tables and associated data used for the examples in this book. You will also learn about various data types and see how to create tables using them. Because the examples in this book are executed against a MySQL database, this chapter is somewhat skewed toward MySQL's features and syntax, but most concepts are applicable to any server.

Creating a MySQL Database

If you want the ability to experiment with the data used for the examples in this book, you have 2 options:

1. Download and install the MySQL server version 8.0 (or later) server and load the Sakila example database from https://dev.mysql.com/doc/index-other.html
2. Go to the page for this book and click on the "Launch Sample Database Session" button.

If you choose the 2nd option, you will have a session available for a period of time (most likely 1 hour), after which any changes you have made to the data will be lost. You may open a new session each time you wish to run SQL statements. This is certainly the easiest option, and I anticipate that most readers will choose this option; if this sounds good to you, feel free to skip ahead to the next section.

If you prefer to have your own copy of the data and want any changes you have made to be permanent, or if you are just interested in installing the MySQL server on your own machine, you may prefer option 1. You may also opt to use a MySQL server hosted in an environment such as Amazon Web Services or Google Cloud. In either case, you will need to perform the installation/configuration yourself, as it is beyond the scope of this book. Once your database is available, you will need to follow a few steps to load the Sakila sample database.

First, you will need to launch the MySQL Command Line Client and provide a password, and then perform the following steps:

1. Go to https://dev.mysql.com/doc/index-other.html and download the files for "sakila database" under the Example Databases section
2. Put the files in a local directory such as C:\temp\sakila-db (used for the next 2 steps, but overwrite with your directory path).
3. Type **`source c:\temp\sakila-db\sakila-schema.sql;`** and press Enter.
4. Type **`source c:\temp\sakila-db\sakila-data.sql;`** and press Enter.

You should now have a working database populated with all the data needed for the examples in this book.

NOTE

The Sakila Sample Database is made available by MySQL, and is licensed via the New BSD license. Sakila contains data for a fictitious movie rental company, and includes tables such as Store, Inventory, Film, Customer, and Payment. While actual movie-rental stores are largely a thing of the past, with a little imagination we could rebrand it as a movie-streaming company by ignoring the Staff and Address tables, and renaming Store to Streaming_Service. However, the examples in this book will stick to the original script (pun intended).

Using the mysql Command-Line Tool

Whether you have your own database server installed or are using a temporary database session (option #2 in the previous section), you will need to start the MySQL command-line tool in order to interact with the database. To do so, you will need to open a Windows or Unix shell and execute the `mysql` utility. For example, if you are logging in using the **root** account, you would do the following:

```
mysql -u root -p;
```

You will then be asked for your password, after which you will see the `mysql>` prompt. To see all of the available databases, you can use the following command:

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
```

```
+--------------------+
5 rows in set (0.01 sec)
```

Since you will be using the Sakila database, you will need to specify the database you want to work with via the **use** command:

```
mysql> use sakila;
Database changed
```

Whenever you invoke the `mysql` command-line tool, you can specify both the username and database to use, as in the following:

```
mysql -u root -p sakila;
```

This will save you from having to type `use sakila;` every time you start up the tool. Now that you have established a session and specified the database, you will be able to issue SQL statements and view the results. For example, if you want to know the current date and time, you could issue the following query:

```
mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2019-04-04 20:44:26 |
+---------------------+
1 row in set (0.01 sec)
```

The `now()` function is a built-in MySQL function that returns the current date and time. As you can see, the `mysql` command-line tool formats the results of your queries within a rectangle bounded by +, -, and | characters. After the results have been exhausted (in this case, there is only a single row of results), the `mysql` command-line tool shows how many rows were returned, along with how long the SQL statement took to execute.

> **ABOUT MISSING FROM CLAUSES**
>
> With some database servers, you won't be able to issue a query without a `from` clause that names at least one table. Oracle Database is a commonly used server for which this is true. For cases when you only need to call a function, Oracle provides a table called `dual`, which consists of a single column called `dummy` that contains a single row of data. In order to be compatible with Oracle Database, MySQL also provides a `dual` table. The previous query to determine the current date and time could therefore be written as:
>
> ```
> mysql> SELECT now()
>        FROM dual;
> +---------------------+
> | now()               |
> +---------------------+
> | 2019-04-04 20:44:26 |
> +---------------------+
> 1 row in set (0.01 sec)
> ```
>
> If you are not using Oracle and have no need to be compatible with Oracle, you can ignore the `dual` table altogether and use just a `select` clause without a `from` clause.

When you are done with the `mysql` command-line tool, simply type **`quit;`** or **`exit;`** to return to the Unix or Windows command shell.

MySQL Data Types

In general, all the popular database servers have the capacity to store the same types of data, such as strings, dates, and numbers. Where they typically differ is in the specialty data types, such as XML and JSON documents or spatial data. Since this is an introductory book on SQL, and since 98% of the columns you encounter will be simple data types, this chapter covers only the character, date (a.k.a.

temporal), and numeric data types. The use of SQL to query JSON documents will be explored in the chapter on SQL and Big Data.

Character Data

Character data can be stored as either fixed-length or variable-length strings; the difference is that fixed-length strings are right-padded with spaces and always consume the same number of bytes, and variable-length strings are not right-padded with spaces and don’t always consume the same number of bytes. When defining a character column, you must specify the maximum size of any string to be stored in the column. For example, if you want to store strings up to 20 characters in length, you could use either of the following definitions:

```
char(20)    /* fixed-length */
varchar(20) /* variable-length */
```

The maximum length for `char` columns is currently 255 bytes, whereas `varchar` columns can be up to 65,535 bytes. If you need to store longer strings (such as emails, XML documents, etc.), then you will want to use one of the text types (`mediumtext` and `longtext`), which I cover later in this section. In general, you should use the `char` type when all strings to be stored in the column are of the same length, such as state abbreviations, and the `varchar` type when strings to be stored in the column are of varying lengths. Both `char` and `varchar` are used in a similar fashion in all the major database servers.

> **NOTE**
>
> Oracle Database is an exception when it comes to the use of `varchar`. Oracle users should use the `varchar2` type when defining variable-length character columns.

CHARACTER SETS

For languages that use the Latin alphabet, such as English, there is a sufficiently small number of characters such that only a single byte is needed to store each character. Other languages, such as Japanese and Korean, contain large numbers of characters, thus requiring multiple bytes of storage for each character. Such character sets are therefore called *multibyte character sets*.

MySQL can store data using various character sets, both single- and multibyte. To view the supported character sets in your server, you can use the `show` command, as in:

```
mysql> SHOW CHARACTER SET;
+----------+---------------------------------+--------------
-------+--------+
| Charset  | Description                     | Default
collation   | Maxlen |
+----------+---------------------------------+--------------
-------+--------+
| armscii8 | ARMSCII-8 Armenian              |
armscii8_general_ci |      1 |
| ascii    | US ASCII                        |
ascii_general_ci    |      1 |
| big5     | Big5 Traditional Chinese        |
big5_chinese_ci     |      2 |
| binary   | Binary pseudo charset           |
binary              |      1 |
| cp1250   | Windows Central European        |
cp1250_general_ci   |      1 |
| cp1251   | Windows Cyrillic                |
```

```
cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic                  |
cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic                  |
cp1257_general_ci   |      1 |
| cp850    | DOS West European               |
cp850_general_ci    |      1 |
| cp852    | DOS Central European            |
cp852_general_ci    |      1 |
| cp866    | DOS Russian                     |
cp866_general_ci    |      1 |
| cp932    | SJIS for Windows Japanese       |
cp932_japanese_ci   |      2 |
| dec8     | DEC West European               |
dec8_swedish_ci     |      1 |
| eucjpms  | UJIS for Windows Japanese       |
eucjpms_japanese_ci |      3 |
| euckr    | EUC-KR Korean                   |
euckr_korean_ci     |      2 |
| gb18030  | China National Standard GB18030 |
gb18030_chinese_ci  |      4 |
| gb2312   | GB2312 Simplified Chinese       |
gb2312_chinese_ci   |      2 |
| gbk      | GBK Simplified Chinese          |
gbk_chinese_ci      |      2 |
| geostd8  | GEOSTD8 Georgian                |
geostd8_general_ci  |      1 |
| greek    | ISO 8859-7 Greek                |
greek_general_ci    |      1 |
| hebrew   | ISO 8859-8 Hebrew               |
hebrew_general_ci   |      1 |
| hp8      | HP West European                |
hp8_english_ci      |      1 |
| keybcs2  | DOS Kamenicky Czech-Slovak      |
keybcs2_general_ci  |      1 |
| koi8r    | KOI8-R Relcom Russian           |
koi8r_general_ci    |      1 |
| koi8u    | KOI8-U Ukrainian                |
koi8u_general_ci    |      1 |
| latin1   | cp1252 West European            |
latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European     |
latin2_general_ci   |      1 |
| latin5   | ISO 8859-9 Turkish              |
latin5_turkish_ci   |      1 |
| latin7   | ISO 8859-13 Baltic              |
latin7_general_ci   |      1 |
| macce    | Mac Central European            |
```

```
macce_general_ci    |      1 |
| macroman | Mac West European               |
macroman_general_ci |      1 |
| sjis     | Shift-JIS Japanese              |
sjis_japanese_ci    |      2 |
| swe7     | 7bit Swedish                    |
swe7_swedish_ci     |      1 |
| tis620   | TIS620 Thai                     |
tis620_thai_ci      |      1 |
| ucs2     | UCS-2 Unicode                   |
ucs2_general_ci     |      2 |
| ujis     | EUC-JP Japanese                 |
ujis_japanese_ci    |      3 |
| utf16    | UTF-16 Unicode                  |
utf16_general_ci    |      4 |
| utf16le  | UTF-16LE Unicode                |
utf16le_general_ci  |      4 |
| utf32    | UTF-32 Unicode                  |
utf32_general_ci    |      4 |
| utf8     | UTF-8 Unicode                   |
utf8_general_ci     |      3 |
| utf8mb4  | UTF-8 Unicode                   |
utf8mb4_0900_ai_ci  |      4 |
+----------+---------------------------------+-------------
-------+--------+
41 rows in set (0.04 sec)
```

If the value in the fourth column, `maxlen`, is greater than 1, then the character set is a multibyte character set.

In prior versions of the MySQL server, the `latin1` character set was automatically chosen as the default character set, but version 8 defaults to **utf8mb4**. However, you may choose to use a different character set for each character column in your database, and you can even store different character sets within the same table. To choose a character set other than the default when defining a column, simply name one of the supported character sets after the type definition, as in:

```
varchar(20) character set latin1
```

With MySQL, you may also set the default character set for your entire database:

```
create database european_sales character set latin1;
```

While this is as much information regarding character sets as is appropriate for an introductory book, there is a great deal more to the topic of internationalization than what is shown here. If you plan to deal with multiple or unfamiliar character sets, you may want to pick up a book such as Jukka Korpela's *Unicode Explained: Internationalize Documents, Programs, and Web Sites* (O'Reilly).

TEXT DATA

If you need to store data that might exceed the 64 KB limit for `varchar` columns, you will need to use one of the text types.

Table 2-1 shows the available text types and their maximum sizes.

Table 2-1. MySQL text types

Text type	Maximum number of bytes
`Tinytext`	255
`Text`	65,535
`Mediumtext`	16,777,215
`Longtext`	4,294,967,295

When choosing to use one of the text types, you should be aware of the following:

- If the data being loaded into a text column exceeds the maximum size for that type, the data will be truncated.
- Trailing spaces will not be removed when data is loaded into the column.
- When using `text` columns for sorting or grouping, only the first 1,024 bytes are used, although this limit may be increased if necessary.

- The different text types are unique to MySQL. SQL Server has a single `text` type for large character data, whereas DB2 and Oracle use a data type called `clob`, for Character Large Object.
- Now that MySQL allows up to 65,535 bytes for `varchar` columns (it was limited to 255 bytes in version 4), there isn't any particular need to use the `tinytext` or `text` type.

If you are creating a column for free-form data entry, such as a `notes` column to hold data about customer interactions with your company's customer service department, then `varchar` will probably be adequate. If you are storing documents, however, you should choose either the `mediumtext` or `longtext` type.

> **NOTE**
>
> Oracle Database allows up to 2,000 bytes for `char` columns and 4,000 bytes for `varchar2` columns. For larger documents you may use the **CLOB** type. SQL Server can handle up to 8,000 bytes for both `char` and `varchar` data, but you can store up to 2GB of data in a column defined as **varchar(max)**.

Numeric Data

Although it might seem reasonable to have a single numeric data type called "numeric," there are actually several different numeric data types that reflect the various ways in which numbers are used, as illustrated here:

A column indicating whether a customer order has been shipped

This type of column, referred to as a *Boolean*, would contain a `0` to indicate `false` and a `1` to indicate `true`.

A system-generated primary key for a transaction table

This data would generally start at `1` and increase in increments of one up to a potentially very large number.

An item number for a customer's electronic shopping basket

The values for this type of column would be positive whole numbers between 1 and, perhaps, 200 (for shopaholics).

Positional data for a circuit board drill machine

High-precision scientific or manufacturing data often requires accuracy to eight decimal points.

To handle these types of data (and more), MySQL has several different numeric data types. The most commonly used numeric types are those used to store whole numbers, or **integers**. When specifying one of these types, you may also specify that the data is *unsigned*, which tells the server that all data stored in the column will be greater than or equal to zero. Table 2-2 shows the five different data types used to store whole-number integers.

Table 2-2. MySQL integer types

Type	Signed range	Unsigned range
`Tinyint`	−128 to 127	0 to 255
`Smallint`	−32,768 to 32,767	0 to 65,535
`Mediumint`	−8,388,608 to 8,388,607	0 to 16,777,215
`Int`	−2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
`Bigint`	−2^63 to 2^63 - 1	0 to 2^64 - 1

When you create a column using one of the integer types, MySQL will allocate an appropriate amount of space to store the data, which ranges from one byte for a `tinyint` to eight bytes for a `bigint`. Therefore, you should try to choose a type that will be large enough to hold the biggest number you can envision being stored in the column without needlessly wasting storage space.

For floating-point numbers (such as 3.1415927), you may choose from the numeric types shown in Table 2-3.

Table 2-3. MySQL floating-point types

Type	Numeric range
`Float(` *`p`* `,` *`s`* `)`	−3.402823466E+38 to −1.175494351E-38 and 1.175494351E-38 to 3.402823466E+38
`Double(` *`p`* `,` *`s`* `)`	−1.7976931348623157E+308 to −2.2250738585072014E-308 and 2.2250738585072014E-308 to 1.7976931348623157E+308

When using a floating-point type, you can specify a *precision* (the total number of allowable digits both to the left and to the right of the decimal point) and a *scale* (the number of allowable digits to the right of the decimal point), but they are not required. These values are represented in Table 2-3 as *`p`* and *`s`*. If you specify a precision and scale for your floating-point column, remember that the data stored in the column will be rounded if the number of digits exceeds the scale and/or precision of the column. For example, a column defined as `float(4,2)` will store a total of four digits, two to the left of the decimal and two to the right of the decimal. Therefore, such a column would handle the numbers 27.44 and 8.19 just fine, but the number 17.8675 would be rounded to 17.87, and attempting to store the number 178.375 in your `float(4,2)` column would generate an error.

Like the integer types, floating-point columns can be defined as `unsigned`, but this designation only prevents negative numbers from being stored in the column rather than altering the range of data that may be stored in the column.

Temporal Data

Along with strings and numbers, you will almost certainly be working with information about dates and/or times. This type of data is referred to as *temporal*, and some examples of temporal data in a database include:

- The future date that a particular event is expected to happen, such as shipping a customer's order
- The date that a customer's order was shipped
- The date and time that a user modified a particular row in a table
- An employee's birth date
- The year corresponding to a row in a `yearly_sales` fact table in a data warehouse
- The elapsed time needed to complete a wiring harness on an automobile assembly line

MySQL includes data types to handle all of these situations. Table 2-4 shows the temporal data types supported by MySQL.

Table 2-4. MySQL temporal types

Type	Default format	Allowable values
`Date`	YYYY-MM-DD	`1000-01-01` to `9999-12-31`
`Datetime`	YYYY-MM-DD HH:MI:SS	`1000-01-01 00:00:00.000000` to `9999-12-31 23:59:59.999999`
`Timestamp`	YYYY-MM-DD HH:MI:SS	`1970-01-01 00:00:00.000000` to `2038-01-18 22:14:07.999999`
`Year`	YYYY	`1901` to `2155`

Type	Default format	Allowable values
`Time`	HHH:MI:SS	`−838:59:59.000000` to `838:59:59.000000`

While database servers store temporal data in various ways, the purpose of a format string (second column of Table 2-4) is to show how the data will be represented when retrieved, along with how a date string should be constructed when inserting or updating a temporal column. Thus, if you wanted to insert the date March 23, 2020 into a `date` column using the default format `YYYY-MM-DD`, you would use the string `'2020-03-23'`.

The **datetime**, **timestamp**, and **time** types also allow fractional seconds of up to 6 decimal places (microseconds). When defining columns using one of these data types, you may supply a value from 0 to 6; for example, specifying datetime(2) would allow your time values to include hundredths of a second.

> **NOTE**
>
> Each database server allows a different range of dates for temporal columns. Oracle Database accepts dates ranging from 4712 BC to 9999 AD, while SQL Server only handles dates ranging from 1753 AD to 9999 AD (unless you are using SQL Server 2008's `datetime2` data type, which allows for dates ranging from 1 AD to 9999 AD). MySQL falls in between Oracle and SQL Server and can store dates from 1000 AD to 9999 AD. Although this might not make any difference for most systems that track current and future events, it is important to keep in mind if you are storing historical dates.

Table 2-5 describes the various components of the date formats shown in Table 2-4.

Table 2-5. Date format components

Component	Definition	Range
YYYY	Year, including century	`1000` to `9999`
MM	Month	`01` (January) to `12` (December)
DD	Day	`01` to `31`
HH	Hour	`00` to `23`
HHH	Hours (elapsed)	`-838` to `838`
MI	Minute	`00` to `59`

Component	Definition	Range
SS	Second	`00` to `59`

Here’s how the various temporal types would be used to implement the examples shown earlier:

- Columns to hold the expected future shipping date of a customer order and an employee’s birth date would use the `date` type, since it is unnecessary to know at what time a person was born and unrealistic to schedule a future shipment down to the second.
- A column to hold information about when a customer order was actually shipped would use the `datetime` type, since it is important to track not only the date that the shipment occurred but the time as well.
- A column that tracks when a user last modified a particular row in a table would use the `timestamp` type. The `timestamp` type holds the same information as the `datetime` type (year, month, day, hour, minute, second), but a `timestamp` column will automatically be populated with the current date/time by the MySQL server when a row is added to a table or when a row is later modified.
- A column holding just year data would use the `year` type.
- Columns that hold data regarding the length of time needed to complete a task would use the `time` type. For this type of data, it would be unnecessary and confusing to store a date component, since you are interested only in the number of

hours/minutes/seconds needed to complete the task. This information could be derived using two `datetime` columns (one for the task start date/time and the other for the task completion date/time) and subtracting one from the other, but it is simpler to use a single `time` column.

Table Creation

Now that you have a firm grasp on what data types may be stored in a MySQL database, it's time to see how to use these types in table definitions. Let's start by defining a table to hold information about a person.

Step 1: Design

A good way to start designing a table is to do a bit of brainstorming to see what kind of information would be helpful to include. Here's what I came up with after thinking for a short time about the types of information that describe a person:

- Name
- Eye color
- Birth date
- Address
- Favorite foods

This is certainly not an exhaustive list, but it's good enough for now. The next step is to assign column names and data types. Table 2-6 shows my initial attempt.

Table 2-6. Person table, first pass

Column	Type	Allowable values
`Name`	`Varchar(40)`	
`Eye_color`	`Char(2)`	`BL`, `BR`, `GR`
`Birth_date`	`Date`	
`Address`	`Varchar(100)`	
`Favorite_foods`	`Varchar(200)`	

The `name`, `address`, and `favorite_foods` columns are of type `varchar` and allow for free-form data entry. The `eye_color` column allows 2 characters which should equal only `BR`, `BL`, or `GR`. The `birth_date` column is of type `date`, since a time component is not needed.

Step 2: Refinement

In Chapter 1, you were introduced to the concept of *normalization*, which is the process of ensuring that there are no duplicate (other than foreign keys) or compound columns in your database design. In looking at the columns in the `person` table a second time, the following issues arise:

- The `name` column is actually a compound object consisting of a first name and a last name.
- Since multiple people can have the same name, eye color, birth date, and so forth, there are no columns in the `person` table that guarantee uniqueness.
- The `address` column is also a compound object consisting of street, city, state/province, country, and postal code.
- The `favorite_foods` column is a list containing `0`, `1`, or more independent items. It would be best to create a separate table for this data that includes a foreign key to the `person` table so that you know to which person a particular food may be attributed.

After taking these issues into consideration, Table 2-7 gives a normalized version of the `person` table.

Table 2-7. Person table, second pass

Column	Type	Allowable values
`Person_id`	`Smallint (unsigned)`	
`First_name`	`Varchar(20)`	
`Last_name`	`Varchar(20)`	
`Eye_color`	`Char(2)`	BR, BL, GR
`Birth_date`	`Date`	
`Street`	`Varchar(30)`	

Column	Type	Allowable values
`City`	`Varchar(20)`	
`State`	`Varchar(20)`	
`Country`	`Varchar(20)`	
`Postal_code`	`Varchar(20)`	

Now that the `person` table has a primary key (`person_id`) to guarantee uniqueness, the next step is to build a `favorite_food` table that includes a foreign key to the `person` table. Table 2-8 shows the result.

Table 2-8. Favorite_food table

Column	Type
`Person_id`	`Smallint (unsigned)`
`Food`	`Varchar(20)`

The `person_id` and `food` columns comprise the primary key of the `favorite_food` table, and the `person_id` column is also a foreign key to the `person` table.

> **HOW MUCH IS ENOUGH?**
>
> Moving the `favorite_foods` column out of the `person` table was definitely a good idea, but are we done yet? What happens, for example, if one person lists “pasta” as a favorite food while another person lists “spaghetti”? Are they the same thing? In order to prevent this problem, you might decide that you want people to choose their favorite foods from a list of options, in which case you should create a `food` table with `food_id` and `food_name` columns, and then change the `favorite_food` table to contain a foreign key to the `food` table. While this design would be fully normalized, you might decide that you simply want to store the values that the user has entered, in which case you may leave the table as is.

Step 3: Building SQL Schema Statements

Now that the design is complete for the two tables holding information about people and their favorite foods, the next step is to generate SQL statements to create the tables in the database. Here is the statement to create the `person` table:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  eye_color CHAR(2),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Everything in this statement should be fairly self-explanatory except for the last item; when you define your table, you need to tell the database server what column or columns will serve as the primary key for the table. You do this by creating a *constraint* on the table. You can add several types of constraints to a table definition. This constraint is a *primary key constraint*. It is created on the `person_id` column and given the name `pk_person`.

While on the topic of constraints, there is another type of constraint that would be useful for the `person` table. In Table 2-6, I added a third column to show the allowable values for certain columns (such as `'BR'` and `'BL'` for the `eye_color` column). Another type of constraint called a *check constraint* constrains the allowable values for a particular column. MySQL allows a check constraint to be attached to a column definition, as in the following:

```
eye_color CHAR(2) CHECK (eye_color IN ('BR','BL','GR')),
```

While check constraints operate as expected on most database servers, the MySQL server allows check constraints to be defined but does not enforce them. However, MySQL does provide another character data type called `enum` that merges the check constraint into the data type definition. Here's what it would look like for the `eye_color` column definition:

```
eye_color ENUM('BR','BL','GR'),
```

Here's how the `person` table definition looks with an `enum` data type for the `eye_color` column:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  eye_color ENUM('BR','BL','GR'),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Later in this chapter, you will see what happens if you try to add data to a column that violates its check constraint (or, in the case of MySQL, its enumeration values).

You are now ready to run the `create table` statement using the `mysql` command-line tool. Here's what it looks like:

```
mysql> CREATE TABLE person
    ->  (person_id SMALLINT UNSIGNED,
    ->   fname VARCHAR(20),
```

```
    ->   lname VARCHAR(20),
    ->   eye_color ENUM('BR','BL','GR'),
    ->   birth_date DATE,
    ->   street VARCHAR(30),
    ->   city VARCHAR(20),
    ->   state VARCHAR(20),
    ->   country VARCHAR(20),
    ->   postal_code VARCHAR(20),
    ->   CONSTRAINT pk_person PRIMARY KEY (person_id)
    ->  );
Query OK, 0 rows affected (0.37 sec)
```

After processing the `create table` statement, the MySQL server returns the message "Query OK, 0 rows affected," which tells me that the statement had no syntax errors. If you want to make sure that the `person` table does, in fact, exist, you can use the `describe` command (or `desc` for short) to look at the table definition:

```
mysql> desc person;
+-------------+----------------------+------+-----+---------
+-------+
| Field       | Type                 | Null | Key | Default
| Extra |
+-------------+----------------------+------+-----+---------
+-------+
| person_id   | smallint(5) unsigned | NO   | PRI | NULL
|       |
| fname       | varchar(20)          | YES  |     | NULL
|       |
| lname       | varchar(20)          | YES  |     | NULL
|       |
| eye_color   | enum('BR','BL','GR') | YES  |     | NULL
|       |
| birth_date  | date                 | YES  |     | NULL
|       |
| street      | varchar(30)          | YES  |     | NULL
|       |
| city        | varchar(20)          | YES  |     | NULL
|       |
| state       | varchar(20)          | YES  |     | NULL
|       |
| country     | varchar(20)          | YES  |     | NULL
|       |
```

```
| postal_code | varchar(20)          | YES  |     | NULL
|       |
+-------------+----------------------+------+-----+---------
+-------+
10 rows in set (0.00 sec)
```

Columns 1 and 2 of the `describe` output are self-explanatory. Column 3 shows whether a particular column can be omitted when data is inserted into the table. I purposefully left this topic out of the discussion for now (see the sidebar "What Is Null?" for a short discourse), but we explore it fully in Chapter 4. The fourth column shows whether a column takes part in any keys (primary or foreign); in this case, the `person_id` column is marked as the primary key. Column 5 shows whether a particular column will be populated with a default value if you omit the column when inserting data into the table. The sixth column (called "Extra") shows any other pertinent information that might apply to a column.

WHAT IS NULL?

In some cases, it is not possible or applicable to provide a value for a particular column in your table. For example, when adding data about a new customer order, the `ship_date` column cannot yet be determined. In this case, the column is said to be *null* (note that I do not say that it *equals* null), which indicates the absence of a value. Null is used for various cases where a value cannot be supplied, such as:

- Not applicable
- Unknown
- Empty set

When designing a table, you may specify which columns are allowed to be null (the default), and which columns are not allowed to be null (designated by adding the keywords `not null` after the type definition).

Now that you've created the `person` table, your next step is to create the `favorite_food` table:

```
mysql> CREATE TABLE favorite_food
    ->  (person_id SMALLINT UNSIGNED,
    ->   food VARCHAR(20),
    ->   CONSTRAINT pk_favorite_food PRIMARY KEY (person_id,
food),
    ->   CONSTRAINT fk_fav_food_person_id FOREIGN KEY
(person_id)
    ->     REFERENCES person (person_id)
    ->  );
Query OK, 0 rows affected (0.10 sec)
```

This should look very similar to the `create table` statement for the `person` table, with the following exceptions:

- Since a person can have more than one favorite food (which is the reason this table was created in the first place), it takes more than just the `person_id` column to guarantee uniqueness in the table. This table, therefore, has a two-column primary key: `person_id` and `food`.
- The `favorite_food` table contains another type of constraint called a *foreign key constraint*. This constrains the values of the `person_id` column in the `favorite_food` table to include *only* values found in the `person` table. With this constraint in place, I will not be able to add a row to the `favorite_food` table indicating that `person_id 27` likes pizza if there isn't already a row in the `person` table having a `person_id` of `27`.

> **NOTE**
>
> If you forget to create the foreign key constraint when you first create the table, you can add it later via the `alter table` statement.

`Describe` shows the following after executing the `create table` statement:

```
mysql> desc favorite_food;
+-----------+----------------------+------+-----+---------+-
------+
| Field     | Type                 | Null | Key | Default |
Extra |
+-----------+----------------------+------+-----+---------+-
------+
| person_id | smallint(5) unsigned | NO   | PRI | NULL
|       |
| food      | varchar(20)          | NO   | PRI | NULL
|       |
+-----------+----------------------+------+-----+---------+-
------+
2 rows in set (0.00 sec)
```

Now that the tables are in place, the next logical step is to add some data.

Populating and Modifying Tables

With the `person` and `favorite_food` tables in place, you can now begin to explore the four SQL data statements: `insert`, `update`, `delete`, and `select`.

Inserting Data

Since there is not yet any data in the `person` and `favorite_food` tables, the first of the four SQL data statements to be explored will be the `insert` statement. There are three main components to an `insert` statement:

- The name of the table into which to add the data
- The names of the columns in the table to be populated
- The values with which to populate the columns

You are not required to provide data for every column in the table (unless all the columns in the table have been defined as `not null`). In some cases, those columns that are not included in the initial `insert` statement will be given a value later via an `update` statement. In other cases, a column may never receive a value for a particular row of data (such as a customer order that is canceled before being shipped, thus rendering the `ship_date` column inapplicable).

GENERATING NUMERIC KEY DATA

Before inserting data into the `person` table, it would be useful to discuss how values are generated for numeric primary keys. Other than picking a number out of thin air, you have a couple of options:

- Look at the largest value currently in the table and add one.
- Let the database server provide the value for you.

Although the first option may seem valid, it proves problematic in a multiuser environment, since two users might look at the table at the same time and generate the same value for the primary key. Instead, all database servers on the market today provide a safe, robust method for generating numeric keys. In some servers, such as the Oracle Database, a separate schema object is used (called a *sequence*); in the case of MySQL, however, you simply need to turn on the *auto-increment* feature for your primary key column. Normally, you would do this at table creation, but doing it now provides the opportunity to learn another SQL schema statement, `alter table`, which is used to modify the definition of an existing table:

```
ALTER TABLE person MODIFY person_id SMALLINT UNSIGNED
AUTO_INCREMENT;
```

This statement essentially redefines the `person_id` column in the `person` table. If you describe the table, you will now see the auto-increment feature listed under the “Extra” column for `person_id`:

```
mysql> DESC person;
+-------------+--------------------------+------+-----+---
------+-----------------+
| Field       | Type                     | Null | Key |
Default | Extra           |
+-------------+--------------------------+------+-----+---
------+-----------------+
| person_id   | smallint(5) unsigned     | NO   | PRI |
NULL    | auto_increment  |
| .           |                          |      |     |
|                 |
| .           |                          |      |     |
|                 |
| .           |                          |      |     |
|                 |
```

When you insert data into the `person` table, simply provide a `null` value for the `person_id` column, and MySQL will populate the column with the next available number (by default, MySQL starts at 1 for auto-increment columns).

THE INSERT STATEMENT

Now that all the pieces are in place, it's time to add some data. The following statement creates a row in the `person` table for William Turner:

```
mysql> INSERT INTO person
    ->   (person_id, fname, lname, eye_color, birth_date)
    -> VALUES (null, 'William','Turner', 'BR', '1972-05-
27');
Query OK, 1 row affected (0.22 sec)
```

The feedback ("Query OK, 1 row affected") tells you that your statement syntax was proper, and that one row was added to the database (since it was an `insert` statement). You can look at the data just added to the table by issuing a `select` statement:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.06 sec)
```

As you can see, the MySQL server generated a value of `1` for the primary key. Since there is only a single row in the `person` table, I neglected to specify which row I am interested in and simply retrieved all the rows in the table. If there were more than one row in the table, however, I could add a `where` clause to specify that I want to retrieve data only for the row having a value of `1` for the `person_id` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE person_id = 1;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

While this query specifies a particular primary key value, you can use any column in the table to search for rows, as shown by the following query, which finds all rows with a value of `'Turner'` for the `lname` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE lname = 'Turner';
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

Before moving on, a couple of things about the earlier `insert` statement are worth mentioning:

- Values were not provided for any of the address columns. This is fine, since `null`s are allowed for those columns.
- The value provided for the `birth_date` column was a string. As long as you match the required format shown in

Table 2-4, MySQL will convert the string to a date for you.

- The column names and the values provided must correspond in number and type. If you name seven columns and provide only six values, or if you provide values that cannot be converted to the appropriate data type for the corresponding column, you will receive an error.

William Turner has also provided information about his favorite three foods, so here are three `insert` statements to store his food preferences:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'pizza');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'cookies');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'nachos');
Query OK, 1 row affected (0.01 sec)
```

Here’s a query that retrieves William’s favorite foods in alphabetical order using an `order by` clause:

```
mysql> SELECT food
    -> FROM favorite_food
    -> WHERE person_id = 1
    -> ORDER BY food;
+---------+
| food    |
+---------+
| cookies |
| nachos  |
| pizza   |
+---------+
3 rows in set (0.02 sec)
```

The `order by` clause tells the server how to sort the data returned by the query. Without the `order by` clause, there is no guarantee that the data in the table will be retrieved in any particular order.

So that William doesn't get lonely, you can execute another `insert` statement to add Susan Smith to the `person` table:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, eye_color, birth_date,
    ->   street, city, state, country, postal_code)
    -> VALUES (null, 'Susan','Smith', 'BL', '1975-11-02',
    ->   '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
 Query OK, 1 row affected (0.01 sec)
```

Since Susan was kind enough to provide her address, we included five more columns than when William's data was inserted. If you query the table again, you will see that Susan's row has been assigned the value `2` for its primary key value:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
|         2 | Susan   | Smith  | 1975-11-02 |
+-----------+---------+--------+------------+
2 rows in set (0.00 sec)
```

CAN I GET THAT IN XML?

If you will be working with XML data, you will be happy to know that most database servers provide a simple way to generate XML output from a query. With MySQL, for example, you can use the `--xml` option when invoking the `mysql` tool, and all your output will automatically be formatted using XML. Here's what the favorite-food data looks like as an XML document:

```
C:\database> mysql -u lrngsql -p --xml bank
Enter password: xxxxxx
Welcome to the MySQL Monitor...

Mysql> SELECT * FROM favorite_food;
<?xml version="1.0"?>

<resultset statement="select * from favorite_food"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="person_id">1</field>
        <field name="food">cookies</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">nachos</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">pizza</field>
  </row>
</resultset>
3 rows in set (0.00 sec)
```

With SQL Server, you don't need to configure your command-line tool; you just need to add the `for xml` clause to the end of your query, as in:

```
SELECT * FROM favorite_food
FOR XML AUTO, ELEMENTS
```

Updating Data

When the data for William Turner was initially added to the table, data for the various address columns was not included the `insert` statement. The next statement shows how these columns can be populated at a later time via an `update` statement:

```
mysql> UPDATE person
    -> SET street = '1225 Tremont St.',
```

```
    ->   city = 'Boston',
    ->   state = 'MA',
    ->   country = 'USA',
    ->   postal_code = '02138'
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The server responded with a two-line message: the “Rows matched: 1” item tells you that the condition in the `where` clause matched a single row in the table, and the “Changed: 1” item tells you that a single row in the table has been modified. Since the `where` clause specifies the primary key of William’s row, this is exactly what you would expect to have happen.

Depending on the conditions in your `where` clause, it is also possible to modify more than one row using a single statement. Consider, for example, what would happen if your `where` clause looked as follows:

```
WHERE person_id < 10
```

Since both William and Susan have a `person_id` value less than 10, both of their rows would be modified. If you leave off the `where` clause altogether, your `update` statement will modify every row in the table.

Deleting Data

It seems that William and Susan aren’t getting along very well together, so one of them has got to go. Since William was there first, Susan will get the boot courtesy of the `delete` statement:

```
mysql> DELETE FROM person
    -> WHERE person_id = 2;
Query OK, 1 row affected (0.01 sec)
```

Again, the primary key is being used to isolate the row of interest, so a single row is deleted from the table. Similar to the `update` statement, more than one row can be deleted depending on the conditions in your `where` clause, and all rows will be deleted if the `where` clause is omitted.

When Good Statements Go Bad

So far, all of the SQL data statements shown in this chapter have been well formed and have played by the rules. Based on the table definitions for the `person` and `favorite_food` tables, however, there are lots of ways that you can run afoul when inserting or modifying data. This section shows you some of the common mistakes that you might come across and how the MySQL server will respond.

Nonunique Primary Key

Because the table definitions include the creation of primary key constraints, MySQL will make sure that duplicate key values are not inserted into the tables. The next statement attempts to bypass the auto-increment feature of the `person_id` column and create another row in the `person` table with a `person_id` of `1`:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, eye_color, birth_date)
    -> VALUES (1, 'Charles','Fulton', 'GR', '1968-01-15');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

There is nothing stopping you (with the current schema objects, at least) from creating two rows with identical names, addresses, birth dates, and so on, as long as they have different values for the `person_id` column.

Nonexistent Foreign Key

The table definition for the `favorite_food` table includes the creation of a foreign key constraint on the `person_id` column. This constraint ensures that all values of `person_id` entered into the `favorite_food` table exist in the `person` table. Here’s what would happen if you tried to create a row that violates this constraint:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (999, 'lasagna');
ERROR 1452 (23000): Cannot add or update a child row: a
foreign key constraint
fails ('bank'.'favorite_food', CONSTRAINT
'fk_fav_food_person_id' FOREIGN KEY
('person_id') REFERENCES 'person' ('person_id'))
```

In this case, the `favorite_food` table is considered the *child* and the `person` table is considered the *parent*, since the `favorite_food` table is dependent on the `person` table for some of its data. If you plan to enter data into both tables, you will need to create a row in `parent` before you can enter data into `favorite_food`.

NOTE

Foreign key constraints are enforced only if your tables are created using the InnoDB storage engine. We discuss MySQL's storage engines in Chapter 12.

Column Value Violations

The `eye_color`column in the `person` table is restricted to the values `'BR'` for brown, `'BL'` for blue, and `'GR'` for green. If you mistakenly attempt to set the value of the column to any other value, you will receive the following response:

```
mysql> UPDATE person
    -> SET eye_color = 'ZZ'
    -> WHERE person_id = 1;
ERROR 1265 (01000): Data truncated for column 'eye_color' at
row 1
```

The error message is a bit confusing, but it gives you the general idea that the server is unhappy about the value provided for the `eye_color` column.

Invalid Date Conversions

If you construct a string with which to populate a `date` column, and that string does not match the expected format, you will receive another error. Here's an example that uses a date format that does not match the default date format of "YYYY-MM-DD":

```
mysql> UPDATE person
    -> SET birth_date = 'DEC-21-1980'
    -> WHERE person_id = 1;
ERROR 1292 (22007): Incorrect date value: 'DEC-21-1980' for
```

```
column 'birth_date'
at row 1
```

In general, it is always a good idea to explicitly specify the format string rather than relying on the default format. Here's another version of the statement that uses the `str_to_date` function to specify which format string to use:

```
mysql> UPDATE person
    -> SET birth_date = str_to_date('DEC-21-1980' , '%b-%d-
%Y')
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Not only is the database server happy, but William is happy as well (we just made him eight years younger, without the need for expensive cosmetic surgery!).

NOTE

Earlier in the chapter, when I discussed the various temporal data types, I showed date-formatting strings such as “YYYY-MM-DD”. While many database servers use this style of formatting, MySQL uses `%Y` to indicate a four-character year. Here are a few more formatters that you might need when converting strings to `datetime`s in MySQL:

```
%a The short weekday name, such as Sun, Mon, ...
%b The short month name, such as Jan, Feb, ...
%c The numeric month (0..12)
%d The numeric day of the month (00..31)
%f The number of microseconds (000000..999999)
%H The hour of the day, in 24-hour format (00..23)
%h The hour of the day, in 12-hour format (01..12)
%i The minutes within the hour (00..59)
%j The day of year (001..366)
%M The full month name (January..December)
%m The numeric month
%p AM or PM
%s The number of seconds (00..59)
%W The full weekday name (Sunday..Saturday)
%w The numeric day of the week (0=Sunday..6=Saturday)
%Y The four-digit year
```

The Sakila Database

For the remainder of the book, most examples will use a sample database called Sakila, which is made available by the nice people at MySQL. This database models a chain of DVD rental stores, which is a bit outdated, but with a bit of imagination it can be rebranded as a video-streaming company. Some of the tables include `Customer`, `Film`, `Actor`, `Payment`, `Rental`, and `Category`. The entire schema and example data should have been created when you followed the final steps at the beginning of the chapter for loading the MySQL server and generating the sample data. To see a diagram of the tables and their columns and relationships, see Appendix A.

Table 2-9 shows some of the tables used in the sakila schema along with short definitions.

Table 2-9. Sakila schema definitions

Table name	Definition
`Film`	A movie which has been released and can be rented
`Actor`	A person who plays acts in films
`Customer`	A person who watches films
`Category`	A genre of films
`Payment`	A rental of a film by a customer
`Language`	A language spoken by the actors of a film

Table name	Definition
`Film_Actor`	An actor in a film
`Inventory`	A film available for rental

Feel free to experiment with the tables as much as you want, including adding your own tables to expand the business functions. You can always drop the database and re-create it from the downloaded file if you want to make sure your sample data is intact. If you are using the temporary session, any changes you make will be lost when the session closes, so you may want to keep a script of your changes so you can recreate any changes you have made.

If you want to see the tables available in your database, you can use the `show tables` command, as in:

```
mysql> show tables;
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
```

```
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.02 sec)
```

Along with the 23 tables in the sakila schema, your table listing may also include the two tables created in this chapter: `person` and `favorite_food`. These tables will not be used in later chapters, so feel free to drop them by issuing the following commands:

```
mysql> DROP TABLE favorite_food;
Query OK, 0 rows affected (0.56 sec)
mysql> DROP TABLE person;
Query OK, 0 rows affected (0.05 sec)
```

If you want to look at the columns in a table, you can use the `describe` command. Here's an example of the `describe` output for the `customer` table:

```
mysql> desc customer;
+-------------+----------------------+------+-----+---------
----------+-------------------------------+
| Field       | Type                 | Null | Key |
Default           | Extra                         |
+-------------+----------------------+------+-----+---------
----------+-------------------------------+
| customer_id | smallint(5) unsigned | NO   | PRI |
NULL              | auto_increment                |
| store_id    | tinyint(3) unsigned  | NO   | MUL |
```

```
NULL              |                               |
| first_name  | varchar(45)          | NO   |     |
NULL              |                               |
| last_name   | varchar(45)          | NO   | MUL |
NULL              |                               |
| email       | varchar(50)          | YES  |     |
NULL              |                               |
| address_id  | smallint(5) unsigned | NO   | MUL |
NULL              |                               |
| active      | tinyint(1)           | NO   |     |
1                 |                               |
| create_date | datetime             | NO   |     |
NULL              |                               |
| last_update | timestamp            | YES  |     |
CURRENT_TIMESTAMP | DEFAULT_GENERATED             |
|
|   on update CURRENT_TIMESTAMP |
+-------------+----------------------+------+-----+---------
----------+-------------------------------+
```

The more comfortable you are with the example database, the better you will understand the examples and, consequently, the concepts in the following chapters.

Chapter 3. Query Primer

So far, you have seen a few examples of database queries (a.k.a. `select` statements) sprinkled throughout the first two chapters. Now it's time to take a closer look at the different parts of the `select` statement and how they interact. After finishing this chapter, you should have a basic understanding of how data is retrieved, joined, filtered, grouped, and sorted; these topics will be covered in detail in chapters 4 through 10.

Query Mechanics

Before dissecting the `select` statement, it might be interesting to look at how queries are executed by the MySQL server (or, for that matter, any database server). If you are using the `mysql` command-line tool (which I assume you are), then you have already logged in to the MySQL server by providing your username and password (and possibly a hostname if the MySQL server is running on a different computer). Once the server has verified that your username and password are correct, a *database connection* is generated for you to use. This connection is held by the application that requested it (which, in this case, is the `mysql` tool) until the application releases the connection (i.e., as a result of your typing `quit`) or the server closes the connection (i.e., when the server is shut down). Each connection to the MySQL server is assigned an identifier, which is shown to you when you first log in:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.15 MySQL Community Server - GPL

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All
rights reserved.
Oracle is a registered trademark of Oracle Corporation
and/or its
affiliates. Other names may be trademarks of their
respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the
buffer.
```

In this case, my connection ID is `11`. This information might be useful to your database administrator if something goes awry, such as a malformed query that runs for hours, so you might want to jot it down.

Once the server has verified your username and password and issued you a connection, you are ready to execute queries (along with other SQL statements). Each time a query is sent to the server, the server checks the following things prior to statement execution:

- Do you have permission to execute the statement?
- Do you have permission to access the desired data?
- Is your statement syntax correct?

If your statement passes these three tests, then your query is handed to the *query optimizer*, whose job it is to determine the most efficient way to execute your query. The optimizer will look at such things as the order in which to join the tables named in your `from` clause and what indexes are available, and then picks an *execution plan*, which the server uses to execute your query.

NOTE

Understanding and influencing how your database server chooses execution plans is a fascinating topic that many of you will wish to explore. For those readers using MySQL, you might consider reading Baron Schwartz et al.'s *High Performance MySQL* (O'Reilly). Among other things, you will learn how to generate indexes, analyze execution plans, influence the optimizer via query hints, and tune your server's startup parameters. If you are using Oracle Database or SQL Server, dozens of tuning books are available.

Once the server has finished executing your query, the *result set* is returned to the calling application (which is, once again, the `mysql` tool). As I mentioned in Chapter 1, a result set is just another table containing rows and columns. If your query fails to yield any results, the `mysql` tool will show you the message found at the end of the following example:

```
mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE last_name = 'ZIEGLER';
Empty set (0.02 sec)
```

If the query returns one or more rows, the `mysql` tool will format the results by adding column headers and by constructing boxes around the columns using the -, |, and + symbols, as shown in the next example:

```
mysql> SELECT *
    -> FROM category;
+-------------+-------------+---------------------+
| category_id | name        | last_update         |
+-------------+-------------+---------------------+
|           1 | Action      | 2006-02-15 04:46:27 |
|           2 | Animation   | 2006-02-15 04:46:27 |
```

```
|           3 | Children    | 2006-02-15 04:46:27 |
|           4 | Classics    | 2006-02-15 04:46:27 |
|           5 | Comedy      | 2006-02-15 04:46:27 |
|           6 | Documentary | 2006-02-15 04:46:27 |
|           7 | Drama       | 2006-02-15 04:46:27 |
|           8 | Family      | 2006-02-15 04:46:27 |
|           9 | Foreign     | 2006-02-15 04:46:27 |
|          10 | Games       | 2006-02-15 04:46:27 |
|          11 | Horror      | 2006-02-15 04:46:27 |
|          12 | Music       | 2006-02-15 04:46:27 |
|          13 | New         | 2006-02-15 04:46:27 |
|          14 | Sci-Fi      | 2006-02-15 04:46:27 |
|          15 | Sports      | 2006-02-15 04:46:27 |
|          16 | Travel      | 2006-02-15 04:46:27 |
+-------------+-------------+---------------------+
16 rows in set (0.02 sec)
```

This query returns all three columns for of all the rows in the `category` table. After the last row of data is displayed, the `mysql` tool displays a message telling you how many rows were returned, which, in this case, is 16.

Query Clauses

Several components or *clauses* make up the `select` statement. While only one of them is mandatory when using MySQL (the `select` clause), you will usually include at least two or three of the six available clauses. Table 3-1 shows the different clauses and their purposes.

Table 3-1. Query clauses

Clause name	Purpose
`Select`	Determines which columns to include in the query's result set
`From`	Identifies the tables from which to retrieve data and how the tables should be joined
`Where`	Filters out unwanted data
`Group by`	Used to group rows together by common column values
`Having`	Filters out unwanted groups
`Order by`	Sorts the rows of the final result set by one or more columns

All of the clauses shown in Table 3-1 are included in the ANSI specification; additionally, several other clauses are unique to MySQL. The following sections delve into the uses of the six major query clauses.

The select Clause

Even though the `select` clause is the first clause of a `select` statement, it is one of the last clauses that the database server evaluates. The reason for this is that before you can determine what to include in the final result set, you need to know all of the possible columns that *could* be included in the final result set. In order to fully understand the role of the `select` clause, therefore, you will need to understand a bit about the `from` clause. Here's a query to get started:

```
mysql> SELECT *
    -> FROM language;
+-------------+----------+---------------------+
| language_id | name     | last_update         |
+-------------+----------+---------------------+
|           1 | English  | 2006-02-15 05:02:19 |
|           2 | Italian  | 2006-02-15 05:02:19 |
|           3 | Japanese | 2006-02-15 05:02:19 |
|           4 | Mandarin | 2006-02-15 05:02:19 |
|           5 | French   | 2006-02-15 05:02:19 |
|           6 | German   | 2006-02-15 05:02:19 |
+-------------+----------+---------------------+
6 rows in set (0.03 sec)
```

In this query, the `from` clause lists a single table (`language`), and the `select` clause indicates that *all* columns (designated by `*`) in the `language` table should be included in the result set. This query could be described in English as follows:

Show me all the columns and all the rows in the `language` *table.*

In addition to specifying all the columns via the asterisk character, you can explicitly name the columns you are interested in, such as:

```
mysql> SELECT language_id, name, last_update
    -> FROM language;
+-------------+----------+---------------------+
| language_id | name     | last_update         |
+-------------+----------+---------------------+
|           1 | English  | 2006-02-15 05:02:19 |
|           2 | Italian  | 2006-02-15 05:02:19 |
|           3 | Japanese | 2006-02-15 05:02:19 |
|           4 | Mandarin | 2006-02-15 05:02:19 |
|           5 | French   | 2006-02-15 05:02:19 |
|           6 | German   | 2006-02-15 05:02:19 |
+-------------+----------+---------------------+
6 rows in set (0.00 sec)
```

The results are identical to the first query, since all the columns in the `language` table (`language_id`, `name`, and `last_update`) are named in the `select` clause. You can choose to include only a subset of the columns in the `language` table as well:

```
mysql> SELECT name
    -> FROM language;
+----------+
| name     |
+----------+
| English  |
| Italian  |
| Japanese |
| Mandarin |
| French   |
| German   |
+----------+
6 rows in set (0.00 sec)
```

The job of the `select` clause, therefore, is the following:

The `select` *clause determines which of all possible columns should be included in the query's result set.*

If you were limited to including only columns from the table or tables named in the `from` clause, things would be rather dull. However, you can spice things up by including in your `select` clause such things as:

- Literals, such as numbers or strings
- Expressions, such as `transaction.amount * −1`
- Built-in function calls, such as `ROUND(transaction.amount, 2)`
- User-defined function calls

The next query demonstrates the use of a table column, a literal, an expression, and a built-in function call in a single query against the `employee` table:

```
mysql> SELECT language_id,
    ->   'COMMON' language_usage,
    ->   language_id * 3.1415927 lang_pi_value,
    ->   upper(name) language_name
    -> FROM language;
+-------------+----------------+---------------+-----------
---+
| language_id | language_usage | lang_pi_value |
language_name |
+-------------+----------------+---------------+-----------
---+
|           1 | COMMON         |     3.1415927 |
ENGLISH       |
|           2 | COMMON         |     6.2831854 |
ITALIAN       |
|           3 | COMMON         |     9.4247781 |
JAPANESE      |
|           4 | COMMON         |    12.5663708 |
```

```
MANDARIN       |
|           5 | COMMON         |    15.7079635 |
FRENCH         |
|           6 | COMMON         |    18.8495562 |
GERMAN         |
+-------------+----------------+---------------+------------
---+
6 rows in set (0.04 sec)
```

We cover expressions and built-in functions in detail later, but I wanted to give you a feel for what kinds of things can be included in the `select` clause. If you only need to execute a built-in function or evaluate a simple expression, you can skip the `from` clause entirely. Here's an example:

```
mysql> SELECT version(),
    ->   user(),
    ->   database();
+-----------+----------------+------------+
| version() | user()         | database() |
+-----------+----------------+------------+
| 8.0.15    | root@localhost | sakila     |
+-----------+----------------+------------+
1 row in set (0.00 sec)
```

Since this query simply calls three built-in functions and doesn't retrieve data from any tables, there is no need for a `from` clause.

Column Aliases

Although the `mysql` tool will generate labels for the columns returned by your queries, you may want to assign your own labels. While you might want to assign a new label to a column from a table (if it is poorly or ambiguously named), you will almost certainly want to assign your own labels to those columns in your result set that are generated by expressions or built-in function calls. You can do so by adding a *column alias* after each element of your `select` clause. Here's the previous query against the `language` table, which included column aliases for three of the columns:

```
mysql> SELECT language_id,
    ->   'COMMON' language_usage,
    ->   language_id * 3.1415927 lang_pi_value,
    ->   upper(name) language_name
    -> FROM language;
+-------------+----------------+---------------+-----------
---+
| language_id | language_usage | lang_pi_value |
language_name |
+-------------+----------------+---------------+-----------
---+
|           1 | COMMON         |     3.1415927 |
ENGLISH       |
|           2 | COMMON         |     6.2831854 |
ITALIAN       |
|           3 | COMMON         |     9.4247781 |
JAPANESE      |
|           4 | COMMON         |    12.5663708 |
MANDARIN      |
|           5 | COMMON         |    15.7079635 |
FRENCH        |
|           6 | COMMON         |    18.8495562 |
GERMAN        |
+-------------+----------------+---------------+-----------
---+
6 rows in set (0.04 sec)
```

If you look at the `select` clause, you can see how the column aliases `language_usage`, `lang_pi_value`, and `language_name` are added after the second, third, and fourth columns. I think you will agree that the output is easier to understand with column aliases in place, and it would be easier to work with programmatically if you were issuing the query from within Java or Python rather than interactively via the `mysql` tool. In order to make your column aliases stand out even more, you also have the option of using the `as` keyword before the alias name, as in:

```
mysql> SELECT language_id,
    ->   'COMMON' AS language_usage,
    ->   language_id * 3.1415927 AS lang_pi_value,
    ->   upper(name) AS language_name
    -> FROM language;
```

Many people feel that including the optional `as` keyword improves readability, although I have chosen not to use it for the examples in this book.

Removing Duplicates

In some cases, a query might return duplicate rows of data. For example, if you were to retrieve the IDs of all actors who appeared in a film, you would see the following:

```
mysql> SELECT actor_id FROM film_actor ORDER BY actor_id;
+----------+
| actor_id |
+----------+
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
...
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
```

```
|      200 |
|      200 |
|      200 |
+----------+
5462 rows in set (0.01 sec)
```

Since some actors appeared in more than one film, you will see the same actor ID multiple times. What you probably want in this case is the *distinct* set of actors, instead of seeing the actor IDs repeated for each film in which they appeared. You can achieve this by adding the keyword `distinct` directly after the `select` keyword, as demonstrated by the following:

```
mysql> SELECT DISTINCT actor_id FROM film_actor ORDER BY
actor_id;
+----------+
| actor_id |
+----------+
|        1 |
|        2 |
|        3 |
|        4 |
|        5 |
|        6 |
|        7 |
|        8 |
|        9 |
|       10 |
...
|      192 |
|      193 |
|      194 |
|      195 |
|      196 |
|      197 |
|      198 |
|      199 |
|      200 |
+----------+
200 rows in set (0.01 sec)
```

The result set now contains 200 rows, one for each distinct actor, rather than 5,462 rows, one for each film appearance by an actor.

> **NOTE**
>
> If you simply want a list of all actors, you can query the Actor table rather than reading through all the rows in Film_Actor and removing duplicates.

If you do not want the server to remove duplicate data, or you are sure there will be no duplicates in your result set, you can specify the `ALL` keyword instead of specifying `DISTINCT`. However, the `ALL` keyword is the default and never needs to be explicitly named, so most programmers do not include `ALL` in their queries.

> **WARNING**
>
> Keep in mind that generating a distinct set of results requires the data to be sorted, which can be time-consuming for large result sets. Don’t fall into the trap of using `DISTINCT` just to be sure there are no duplicates; instead, take the time to understand the data you are working with so that you will know whether duplicates are possible.

The from Clause

Thus far, you have seen queries whose `from` clauses contain a single table. Although most SQL books will define the `from` clause as simply a list of one or more tables, I would like to broaden the definition as follows:

The `from` clause defines the tables used by a query, along with the means of linking the tables together.

This definition is composed of two separate but related concepts, which we explore in the following sections.

Tables

When confronted with the term *table*, most people think of a set of related rows stored in a database. While this does describe one type of table, I would like to use the word in a more general way by removing any notion of how the data might be stored and concentrating on just the set of related rows. Three different types of tables meet this relaxed definition:

- Permanent tables (i.e., created using the `create table` statement)
- Derived tables (i.e., rows returned by a subquery and held in memory)
- Temporary tables (i.e. volatile data held in memory)
- Virtual tables (i.e., created using the `create view` statement)

Each of these table types may be included in a query's `from` clause. By now, you should be comfortable with including a permanent table in a `from` clause, so I will briefly describe the other types of tables that can be referenced in a `from` clause.

DERIVED (SUBQUERY-GENERATED) TABLES

A subquery is a query contained within another query. Subqueries are surrounded by parentheses and can be found in various parts of a `select` statement; within the `from` clause, however, a subquery serves the role of generating a derived table that is visible from all other query clauses and can interact with other tables named in the `from` clause. Here's a simple example:

```
mysql> SELECT concat(cust.last_name, ', ', cust.first_name)
full_name
    -> FROM
    ->  (SELECT first_name, last_name, email
    ->   FROM customer
    ->   WHERE first_name = 'JESSIE'
    ->  ) cust;
+---------------+
| full_name     |
+---------------+
| BANKS, JESSIE |
| MILAM, JESSIE |
+---------------+
2 rows in set (0.00 sec)
```

In this example, a subquery against the `employee` table returns three columns, and the *containing query* references two of the three available columns. The subquery is referenced by the containing query via its alias, which, in this case, is `cust`. The data in `cust` is held in memory for the duration of the query and is then discarded. This is a simplistic and not particularly useful example of a subquery in a `from` clause; you will find detailed coverage of subqueries in Chapter 9.

TEMPORARY TABLES

Although the implementations differ, every relational database allows the ability to define volatile, or temporary, tables. These tables look just like permanent tables, but any data inserted into a temporary table will disappear at some point (generally at the end of a transaction or when your database session is closed). Here's a simple example showing how actors whose last names start with J can be created:

```
mysql> CREATE TEMPORARY TABLE actors_j
    ->  (actor_id smallint(5),
    ->   first_name varchar(45),
    ->   last_name varchar(45)
    ->  );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO actors_j
    -> SELECT actor_id, first_name, last_name
    -> FROM actor
    -> WHERE last_name LIKE 'J%';
Query OK, 7 rows affected (0.03 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM actors_j;
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
|      119 | WARREN     | JACKMAN   |
|      131 | JANE       | JACKMAN   |
|        8 | MATTHEW    | JOHANSSON |
|       64 | RAY        | JOHANSSON |
|      146 | ALBERT     | JOHANSSON |
|       82 | WOODY      | JOLIE     |
|       43 | KIRK       | JOVOVICH  |
+----------+------------+-----------+
7 rows in set (0.00 sec)
```

These seven rows are held in memory temporarily and will disappear after your session is closed.

> **NOTE**
>
> Most database servers also drop the temporary table when the session ends. The exception is Oracle Database, which keeps the definition of the temporary table available for future sessions.

VIEWS

A view is a query that is stored in the data dictionary. It looks and acts like a table, but there is no data associated with a view (this is why I call it a *virtual* table). When you issue a query against a view, your query is merged with the view definition to create a final query to be executed.

To demonstrate, here's a view definition that queries the `employee` table and includes a call to a built-in function:

```
mysql> CREATE VIEW cust_vw AS
    -> SELECT customer_id, first_name, last_name, active
    -> FROM customer;
Query OK, 0 rows affected (0.12 sec)
```

When the view is created, no additional data is generated or stored: the server simply tucks away the `select` statement for future use. Now that the view exists, you can issue queries against it, as in:

```
mysql> SELECT first_name, last_name
    -> FROM cust_vw
    -> WHERE active = 0;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| SANDRA     | MARTIN    |
| JUDITH     | COX       |
| SHEILA     | WELLS     |
```

```
| ERICA       | MATTHEWS  |
| HEIDI       | LARSON    |
| PENNY       | NEAL      |
| KENNETH     | GOODEN    |
| HARRY       | ARCE      |
| NATHAN      | RUNYON    |
| THEODORE    | CULP      |
| MAURICE     | CRAWLEY   |
| BEN         | EASTER    |
| CHRISTIAN   | JUNG      |
| JIMMIE      | EGGLESTON |
| TERRANCE    | ROUSH     |
+------------+-----------+
15 rows in set (0.00 sec)
```

Views are created for various reasons, including to hide columns from users and to simplify complex database designs.

Table Links

The second deviation from the simple `from` clause definition is the mandate that if more than one table appears in the `from` clause, the conditions used to *link* the tables must be included as well. This is not a requirement of MySQL or any other database server, but it is the ANSI-approved method of joining multiple tables, and it is the most portable across the various database servers. We explore joining multiple tables in depth in Chapters Chapter 5 and Chapter 10, but here's a simple example in case I have piqued your curiosity:

```
mysql> SELECT customer.first_name, customer.last_name,
    ->   time(rental.rental_date) rental_time
    -> FROM customer
    ->   INNER JOIN rental
    ->   ON customer.customer_id = rental.customer_id
    -> WHERE date(rental.rental_date) = '2005-06-14';
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEFFERY    | PINSON    | 22:53:33    |
```

```
| ELMER      | NOE       | 22:55:13    |
| MINNIE     | ROMERO    | 23:00:34    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| DANIEL     | CABRAL    | 23:09:38    |
| TERRANCE   | ROUSH     | 23:12:46    |
| JOYCE      | EDWARDS   | 23:16:26    |
| GWENDOLYN  | MAY       | 23:16:27    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| MATTHEW    | MAHAN     | 23:25:58    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| SONIA      | GREGORY   | 23:50:11    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| JEANETTE   | GREENE    | 23:54:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

The previous query displays data from both the `customer` table (`first_name`, `last_name`) and the `rental` table (`rental_date`), so both tables are included in the `from` clause. The mechanism for linking the two tables (referred to as a *join*) is the customer ID stored in both the `customer` and `rental` tables. Thus, the database server is instructed to use the value of the `customer_id` column in the `customer` table to find all of the customer's rentals in the `rental` table. Join conditions for the two tables are found in the `on` subclause of the `from` clause; in this case, the join condition is `ON customer.customer_id = rental.customer_id`. The `where` clause is not part of the join, and is only included to keep the result set fairly small, since there are over 16,000 rows in the `rental` table. Again, please refer to Chapter 5 for a thorough discussion of joining multiple tables.

Defining Table Aliases

When multiple tables are joined in a single query, you need a way to identify which table you are referring to when you reference columns in the `select`, `where`, `group by`, `having`, and `order by` clauses. You have two choices when referencing a table outside the `from` clause:

- Use the entire table name, such as `employee.emp_id`.
- Assign each table an *alias* and use the alias throughout the query.

In the previous query, I chose to use the entire table name in the `select` and `on` clauses. Here's what the same query looks like using table aliases:

```
SELECT c.first_name, c.last_name,
  time(r.rental_date) rental_time
FROM customer c
  INNER JOIN rental r
  ON c.customer_id = r.customer_id
WHERE date(r.rental_date) = '2005-06-14';
```

If you look closely at the `from` clause, you will see that the `customer` table is assigned the alias `c`, and the `rental` table is assigned the alias `r`. These aliases are then used in the `on` clause when defining the join condition as well as in the `select` clause when specifying the columns to include in the result set. I hope you will agree that using aliases makes for a more compact statement without causing confusion (as long as your choices for alias names are reasonable). Additionally, you may use the `as` keyword with your table aliases, similar to what was demonstrated earlier for column aliases:

```
SELECT c.first_name, c.last_name,
  time(r.rental_date) rental_time
FROM customer AS c
  INNER JOIN rental AS r
  ON c.customer_id = r.customer_id
WHERE date(r.rental_date) = '2005-06-14';
```

I have found that roughly half of the database developers I have worked with use the `as` keyword with their column and table aliases, and half do not.

The where Clause

In some cases, you may want to retrieve all rows from a table, especially for small tables such as `language`. Most of the time, however, you will not wish to retrieve *every* row from a table but will want a way to filter out those rows that are not of interest. This is a job for the `where` clause.

The `where` *clause is the mechanism for filtering out unwanted rows from your result set.*

For example, perhaps you are interested in renting a film, but you are only interested in movies rated G that can be kept for at least a week. The following query employs a `where` clause to retrieve *only* the films meeting these criteria:

```
mysql> SELECT title
    -> FROM film
    -> WHERE rating = 'G' AND rental_duration >= 7;
+-------------------------+
| title                   |
+-------------------------+
| BLANKET BEVERLY         |
```

```
| BORROWERS BEDAZZLED     |
| BRIDE INTRIGUE          |
| CATCH AMISTAD           |
| CITIZEN SHREK           |
| COLDBLOODED DARLING     |
| CONTROL ANTHEM          |
| CRUELTY UNFORGIVEN      |
| DARN FORRESTER          |
| DESPERATE TRAINSPOTTING |
| DIARY PANIC             |
| DRACULA CRYSTAL         |
| EMPIRE MALKOVICH        |
| FIREHOUSE VIETNAM       |
| GILBERT PELICAN         |
| GRADUATE LORD           |
| GREASE YOUTH            |
| GUN BONNIE              |
| HOOK CHARIOTS           |
| MARRIED GO              |
| MENAGERIE RUSHMORE      |
| MUSCLE BRIGHT           |
| OPERATION OPERATION     |
| PRIMARY GLASS           |
| REBEL AIRPORT           |
| SPIKING ELEMENT         |
| TRUMAN CRAZY            |
| WAKE JAWS               |
| WAR NOTTING             |
+-------------------------+
29 rows in set (0.00 sec)
```

In this case, the `where` clause filtered out 971 of the 1000 rows in the `film` table. This `where` clause contains two *filter conditions*, but you can include as many conditions as required; individual conditions are separated using operators such as `and`, `or`, and `not` (see Chapter 4 for a complete discussion of the `where` clause and filter conditions).

Let’s see what would happen if you change the operator separating the two conditions from `and` to `or`:

```
mysql> SELECT title
    -> FROM film
    -> WHERE rating = 'G' OR rental_duration >= 7;
+-----------------------------+
| title                       |
+-----------------------------+
| ACE GOLDFINGER              |
| ADAPTATION HOLES            |
| AFFAIR PREJUDICE            |
| AFRICAN EGG                 |
| ALAMO VIDEOTAPE             |
| AMISTAD MIDSUMMER           |
| ANGELS LIFE                 |
| ANNIE IDENTITY              |
|...                          |
| WATERSHIP FRONTIER          |
| WEREWOLF LOLA               |
| WEST LION                   |
| WESTWARD SEABISCUIT         |
| WOLVES DESIRE               |
| WON DARES                   |
| WORKER TARZAN               |
| YOUNG LANGUAGE              |
+-----------------------------+
340 rows in set (0.00 sec)
```

When you separate conditions using the `and` operator, *all* conditions must evaluate to `true` to be included in the result set; when you use `or`, however, only *one* of the conditions needs to evaluate to `true` for a row to be included, which explains why the size of the result set has jumped from 29 to 340 rows.

So, what should you do if you need to use both `and` and `or` operators in your `where` clause? Glad you asked. You should use parentheses to group conditions together. The next query specifies that only those films which are rated G and are available for 7 or more days, or are rated PG-13 and are available 3 or fewer days be included in the result set:

```
mysql> SELECT title, rating, rental_duration
    -> FROM film
    -> WHERE (rating = 'G' AND rental_duration >= 7)
    ->   OR (rating = 'PG-13' AND rental_duration < 4);
+-------------------------+--------+-----------------+
| title                   | rating | rental_duration |
+-------------------------+--------+-----------------+
| ALABAMA DEVIL           | PG-13  |               3 |
| BACKLASH UNDEFEATED     | PG-13  |               3 |
| BILKO ANONYMOUS         | PG-13  |               3 |
| BLANKET BEVERLY         | G      |               7 |
| BORROWERS BEDAZZLED     | G      |               7 |
| BRIDE INTRIGUE          | G      |               7 |
| CASPER DRAGONFLY        | PG-13  |               3 |
| CATCH AMISTAD           | G      |               7 |
| CITIZEN SHREK           | G      |               7 |
| COLDBLOODED DARLING     | G      |               7 |
|...                                                  |
| TREASURE COMMAND        | PG-13  |               3 |
| TRUMAN CRAZY            | G      |               7 |
| WAIT CIDER              | PG-13  |               3 |
| WAKE JAWS               | G      |               7 |
| WAR NOTTING             | G      |               7 |
| WORLD LEATHERNECKS      | PG-13  |               3 |
+-------------------------+--------+-----------------+
68 rows in set (0.00 sec)
```

You should always use parentheses to separate groups of conditions when mixing different operators so that you, the database server, and anyone who comes along later to modify your code will be on the same page.

The group by and having Clauses

All the queries thus far have retrieved raw data without any manipulation. Sometimes, however, you will want to find trends in your data that will require the database server to cook the data a bit before you retrieve your result set. One such mechanism is the `group by` clause, which is used to group data by column values.

For example, let’s say you wanted to find all of the customers who have rented 40 or more films. Rather than looking through all 16,044 rows in the `rental` table, you can write a query which instructs the server to group all rentals by customer, count the number of rentals for each customer, and then return only those customers whose rental count is at least 40. When using the `group by` clause to generate groups of rows, you may also use the `having` clause, which allows you to filter grouped data in the same way the `where` clause lets you filter raw data.

Here’s what the query looks like:

```
mysql> SELECT c.first_name, c.last_name, count(*)
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> GROUP BY c.first_name, c.last_name
    -> HAVING count(*) >= 40;
+------------+-----------+----------+
| first_name | last_name | count(*) |
+------------+-----------+----------+
| TAMMY      | SANDERS   |       41 |
| CLARA      | SHAW      |       42 |
| ELEANOR    | HUNT      |       46 |
| SUE        | PETERS    |       40 |
| MARCIA     | DEAN      |       42 |
| WESLEY     | BULL      |       40 |
| KARL       | SEAL      |       45 |
+------------+-----------+----------+
7 rows in set (0.03 sec)
```

I wanted to briefly mention these two clauses so that they don’t catch you by surprise later in the book, but they are a bit more advanced than the other four `select` clauses. Therefore, I ask that you wait until Chapter 8 for a full description of how and when to use `group by` and `having`.

The order by Clause

In general, the rows in a result set returned from a query are not in any particular order. If you want your result set to be sorted, you will need to instruct the server to sort the results using the `order by` clause:

The `order by` *clause is the mechanism for sorting your result set using either raw column data or expressions based on column data.*

For example, here's another look at an earlier query which returns all customers who rented a film on June 14th, 2005:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14';
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEFFERY    | PINSON    | 22:53:33    |
| ELMER      | NOE       | 22:55:13    |
| MINNIE     | ROMERO    | 23:00:34    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| DANIEL     | CABRAL    | 23:09:38    |
| TERRANCE   | ROUSH     | 23:12:46    |
| JOYCE      | EDWARDS   | 23:16:26    |
| GWENDOLYN  | MAY       | 23:16:27    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| MATTHEW    | MAHAN     | 23:25:58    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| SONIA      | GREGORY   | 23:50:11    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| JEANETTE   | GREENE    | 23:54:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

If you would like the results to be in alphabetical order by last name, you can add the `last_name` column to the `order by` clause:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY c.last_name;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| DANIEL     | CABRAL    | 23:09:38    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| JOYCE      | EDWARDS   | 23:16:26    |
| JEANETTE   | GREENE    | 23:54:46    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| MATTHEW    | MAHAN     | 23:25:58    |
| GWENDOLYN  | MAY       | 23:16:27    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
| MINNIE     | ROMERO    | 23:00:34    |
| TERRANCE   | ROUSH     | 23:12:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

While it is not the case in this example, large customer lists will often contain multiple people having the same last name, so you may want to extend the sort criteria to include the person’s first name as well; you can accomplish this by adding the `first_name` column after the `last_name` column in the `order by` clause:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
```

```
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY c.last_name, c.first_name;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| DANIEL     | CABRAL    | 23:09:38    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| JOYCE      | EDWARDS   | 23:16:26    |
| JEANETTE   | GREENE    | 23:54:46    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| MATTHEW    | MAHAN     | 23:25:58    |
| GWENDOLYN  | MAY       | 23:16:27    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
| MINNIE     | ROMERO    | 23:00:34    |
| TERRANCE   | ROUSH     | 23:12:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

The order in which columns appear in your `order by` clause does make a difference when you include more than one column. If you were to switch the order of the two columns in the `order by` clause, Amber Dixon would appear first in the result set.

Ascending Versus Descending Sort Order

When sorting, you have the option of specifying *ascending* or *descending* order via the `asc` and `desc` keywords. The default is ascending, so you will need to add the `desc` keyword, only if you want to use a descending sort. For example, the following query shows all customers who rented films on June 14th 2005 in descending order of rental time:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY time(r.rental_date) desc;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEANETTE   | GREENE    | 23:54:46    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| AMBER      | DIXON     | 23:42:56    |
| HERMAN     | DEVORE    | 23:35:09    |
| MATTHEW    | MAHAN     | 23:25:58    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| GWENDOLYN  | MAY       | 23:16:27    |
| JOYCE      | EDWARDS   | 23:16:26    |
| TERRANCE   | ROUSH     | 23:12:46    |
| DANIEL     | CABRAL    | 23:09:38    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| MINNIE     | ROMERO    | 23:00:34    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

Descending sorts are commonly used for ranking queries, such as “show me the top five account balances.” MySQL includes a `limit` clause that allows you to sort your data and then discard all but the first *X* rows.

Sorting via Numeric Placeholders

If you are sorting using the columns in your `select` clause, you can opt to reference the columns by their *position* in the `select` clause rather than by name. This can be especially helpful if you are sorting on an expression, such as in the previous example. Here’s the previous example one last time, with an `order by` clause specifying a descending sort using the 3rd element in the `select` clause:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY 3 desc;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEANETTE   | GREENE    | 23:54:46    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| AMBER      | DIXON     | 23:42:56    |
| HERMAN     | DEVORE    | 23:35:09    |
| MATTHEW    | MAHAN     | 23:25:58    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| GWENDOLYN  | MAY       | 23:16:27    |
| JOYCE      | EDWARDS   | 23:16:26    |
| TERRANCE   | ROUSH     | 23:12:46    |
| DANIEL     | CABRAL    | 23:09:38    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| MINNIE     | ROMERO    | 23:00:34    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

You might want to use this feature sparingly, since adding a column to the `select` clause without changing the numbers in the `order by` clause can lead to unexpected results. Personally, I may reference columns positionally when writing ad hoc queries, but I always reference columns by name when writing code.

Test Your Knowledge

The following exercises are designed to strengthen your understanding of the `select` statement and its various clauses. Please see Appendix C.

Exercise 3-1

Retrieve the actor ID, first name, and last name for all actors. Sort by last name and then by first name.

Exercise 3-2

Retrieve the actor ID, first name, and last name for all actors whose last name equals `'WILLIAMS'` or .`'DAVIS'`

Exercise 3-3

Write a query against the `rental` table that returns the IDs of the customers who rented a film on July 5th 2005 (use the `rental.rental_date` column, and you can use the `date()` function to ignore the time component). Include a single row for each distinct customer ID.

Exercise 3-4

Fill in the blanks (denoted by <#>) for this multi-table query to achieve the results shown below.

```
mysql> SELECT c.email, r.return_date
    -> FROM customer c
    ->   INNER JOIN rental <1>
    ->   ON c.customer_id = <2>
```

```
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY <3>, <4>;
+----------------------------------------+-------------------
--+
| email                                  |
return_date         |
+----------------------------------------+-------------------
--+
| DANIEL.CABRAL@sakilacustomer.org       | 2005-06-23
22:00:38 |
| TERRANCE.ROUSH@sakilacustomer.org      | 2005-06-23
21:53:46 |
| MIRIAM.MCKINNEY@sakilacustomer.org     | 2005-06-21
17:12:08 |
| GWENDOLYN.MAY@sakilacustomer.org       | 2005-06-20
02:40:27 |
| JEANETTE.GREENE@sakilacustomer.org     | 2005-06-19
23:26:46 |
| HERMAN.DEVORE@sakilacustomer.org       | 2005-06-19
03:20:09 |
| JEFFERY.PINSON@sakilacustomer.org      | 2005-06-18
21:37:33 |
| MATTHEW.MAHAN@sakilacustomer.org       | 2005-06-18
05:18:58 |
| MINNIE.ROMERO@sakilacustomer.org       | 2005-06-18
01:58:34 |
| SONIA.GREGORY@sakilacustomer.org       | 2005-06-17
21:44:11 |
| TERRENCE.GUNDERSON@sakilacustomer.org  | 2005-06-17
05:28:35 |
| ELMER.NOE@sakilacustomer.org           | 2005-06-17
02:11:13 |
| JOYCE.EDWARDS@sakilacustomer.org       | 2005-06-16
21:00:26 |
| AMBER.DIXON@sakilacustomer.org         | 2005-06-16
04:02:56 |
| CHARLES.KOWALSKI@sakilacustomer.org    | 2005-06-16
02:26:34 |
| CATHERINE.CAMPBELL@sakilacustomer.org  | 2005-06-15
20:43:03 |
+----------------------------------------+-------------------
--+
16 rows in set (0.03 sec)
```

Chapter 4. Filtering

Sometimes you will want to work with every row in a table, such as:

- Purging all data from a table used to stage new data warehouse feeds
- Modifying all rows in a table after a new column has been added
- Retrieving all rows from a message queue table

In cases like these, your SQL statements won't need to have a `where` clause, since you don't need to exclude any rows from consideration. Most of the time, however, you will want to narrow your focus to a subset of a table's rows. Therefore, all the SQL data statements (except the `insert` statement) include an optional `where` clause containing one or more *filter conditions* used to restrict the number of rows acted on by the SQL statement. Additionally, the `select` statement includes a `having` clause in which filter conditions pertaining to grouped data may be included. This chapter explores the various types of filter conditions that you can employ in the `where` clauses of `select`, `update`, and `delete` statements; I demonstrate the use of filter conditions in the `having` clause of a `select` statement in Chapter 8.

Condition Evaluation

A `where` clause may contain one or more *conditions*, separated by the operators `and` and `or`. If multiple conditions are separated only by the `and` operator, then all the conditions must evaluate to `true` for the row to be included in the result set. Consider the following `where` clause:

```
WHERE first_name = 'STEVEN' AND create_date > '2006-01-01'
```

Given these two conditions, only rows where the first name is Steven and the creation date was after January 1st 2006 will be included in the result set. Although this example uses only two conditions, no matter how many conditions are in your `where` clause, if they are separated by the `and` operator they must *all* evaluate to `true` for the row to be included in the result set.

If all conditions in the `where` clause are separated by the `or` operator, however, only *one* of the conditions must evaluate to `true` for the row to be included in the result set. Consider the following two conditions:

```
WHERE first_name = 'STEVEN' OR create_date > '2006-01-01'
```

There are now various ways for a given row to be included in the result set:

- The first name is Steven and the creation date was after January 1, 2006
- The first name is Steven and the creation date was on or before January 1, 2006

- The first name is anything other than Steven but the creation date was after January 1, 2006

Table 4-1 shows the possible outcomes for a `where` clause containing two conditions separated by the `or` operator.

Table 4-1. Two-condition evaluation using or

Intermediate result	Final result
`WHERE true OR true`	`True`
`WHERE true OR false`	`True`
`WHERE false OR true`	`True`
`WHERE false OR false`	`False`

In the case of the preceding example, the only way for a row to be excluded from the result set is if the person's first name was not Steven and the creation date was on or before January 1, 2006.

Using Parentheses

If your `where` clause includes three or more conditions using both the `and` and `or` operators, you should use parentheses to make your intent clear, both to the database server and to anyone else reading your code. Here's a `where` clause that extends the previous example by checking to make sure that the first name is Steven or the last name is Young, and the creation date is after January 1st 2006:

```
WHERE (first_name = 'STEVEN' OR last_name = 'YOUNG')
  AND create_date > '2006-01-01'
```

There are now three conditions; for a row to make it to the final result set, either the first *or* second conditions (or both) must evaluate to `true` , and the third condition must evaluate to `true` . Table 4-2 shows the possible outcomes for this `where` clause.

Table 4-2. Three-condition evaluation using and, or

Intermediate result	Final result
`WHERE (true OR true) AND true`	`True`
`WHERE (true OR false) AND true`	`True`
`WHERE (false OR true) AND true`	`True`
`WHERE (false OR false) AND true`	`False`
`WHERE (true OR true) AND false`	`False`
`WHERE (true OR false) AND false`	`False`

Intermediate result	Final result
`WHERE (false OR true) AND false`	`False`
`WHERE (false OR false) AND false`	`False`

As you can see, the more conditions you have in your `where` clause, the more combinations there are for the server to evaluate. In this case, only three of the eight combinations yield a final result of `true`.

Using the not Operator

Hopefully, the previous three-condition example is fairly easy to understand. Consider the following modification, however:

```
WHERE NOT (first_name = 'STEVEN' OR last_name = 'YOUNG')
  AND create_date > '2006-01-01'
```

Did you spot the change from the previous example? I added the `not` operator before the first set of conditions. Now, instead of looking for people with the first name of Steven or the last name of Young whose record was created after January 1st 2006, I am retrieving only rows where the first name is not Steven or the last name is not Young whose record was created after January 1st 2006. Table 4-3 shows the possible outcomes for this example.

Table 4-3. Three-condition evaluation using and, or, and not

Intermediate result	Final result
`WHERE NOT (true OR true) AND true`	`False`
`WHERE NOT (true OR false) AND true`	`False`
`WHERE NOT (false OR true) AND true`	`False`
`WHERE NOT (false OR false) AND true`	`True`
`WHERE NOT (true OR true) AND false`	`False`
`WHERE NOT (true OR false) AND false`	`False`

Intermediate result	Final result
`WHERE NOT (false OR true) AND false`	`False`
`WHERE NOT (false OR false) AND false`	`False`

While it is easy for the database server to handle, it is typically difficult for a person to evaluate a `where` clause that includes the `not` operator, which is why you won't encounter it very often. In this case, you can rewrite the `where` clause to avoid using the `not` operator:

```
WHERE first_name <> 'STEVEN' AND last_name <> 'YOUNG'
  AND create_date > '2006-01-01'
```

While I'm sure that the server doesn't have a preference, you probably have an easier time understanding this version of the `where` clause.

Building a Condition

Now that you have seen how the server evaluates multiple conditions, let's take a step back and look at what comprises a single condition. A condition is made up of one or more *expressions* combined with one or more *operators*. An expression can be any of the following:

- A number
- A column in a table or view
- A string literal, such as `'Maple Street'`
- A built-in function, such as `concat('Learning', ' ', 'SQL')`
- A subquery
- A list of expressions, such as `('Boston', 'New York', 'Chicago')`

The operators used within conditions include:

- Comparison operators, such as `=`, `!=`, `<`, `>`, `<>`, `LIKE`, `IN`, and `BETWEEN`
- Arithmetic operators, such as `+`, `−`, `*`, and `/`

The following section demonstrates how you can combine these expressions and operators to manufacture the various types of conditions.

Condition Types

There are many different ways to filter out unwanted data. You can look for specific values, sets of values, or ranges of values to include or exclude, or you can use various pattern-searching techniques to look for partial matches when dealing with string data. The next four subsections explore each of these condition types in detail.

Equality Conditions

A large percentage of the filter conditions that you write or come across will be of the form `'column = expression'` as in:

```
title = 'RIVER OUTLAW'
fed_id = '111-11-1111'
amount = 375.25
film_id = (SELECT film_id FROM film WHERE title = 'RIVER
OUTLAW')
```

Conditions such as these are called *equality conditions* because they equate one expression to another. The first three examples equate a column to a literal (two strings and a number), and the fourth example equates a column to the value returned from a subquery. The following query uses two equality conditions; one in the `on` clause (a join condition), and the other in the `where` clause (a filter condition):

```
mysql> SELECT c.email
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14';
+---------------------------------------+
| email                                 |
+---------------------------------------+
| CATHERINE.CAMPBELL@sakilacustomer.org |
| JOYCE.EDWARDS@sakilacustomer.org      |
| AMBER.DIXON@sakilacustomer.org        |
| JEANETTE.GREENE@sakilacustomer.org    |
| MINNIE.ROMERO@sakilacustomer.org      |
| GWENDOLYN.MAY@sakilacustomer.org      |
| SONIA.GREGORY@sakilacustomer.org      |
| MIRIAM.MCKINNEY@sakilacustomer.org    |
| CHARLES.KOWALSKI@sakilacustomer.org   |
| DANIEL.CABRAL@sakilacustomer.org      |
| MATTHEW.MAHAN@sakilacustomer.org      |
| JEFFERY.PINSON@sakilacustomer.org     |
| HERMAN.DEVORE@sakilacustomer.org      |
| ELMER.NOE@sakilacustomer.org          |
| TERRANCE.ROUSH@sakilacustomer.org     |
| TERRENCE.GUNDERSON@sakilacustomer.org |
```

```
+--------------------------------------+
16 rows in set (0.03 sec)
```

This query shows all email addresses of every customer who rented a film on June 14 2005.

INEQUALITY CONDITIONS

Another fairly common type of condition is the *inequality condition*, which asserts that two expressions are *not* equal. Here's the previous query with the filter condition in the `where` clause changed to an inequality condition:

```
mysql> SELECT c.email
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) <> '2005-06-14';

+-----------------------------------+
| email                             |
+-----------------------------------+
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
| MARY.SMITH@sakilacustomer.org     |
...
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
| AUSTIN.CINTRON@sakilacustomer.org |
+-----------------------------------+
```

```
16028 rows in set (0.03 sec)
```

This query returns all email addresses for films rented on any other date than June 14 2005. When building inequality conditions, you may choose to use either the `!=` or `<>` operator.

DATA MODIFICATION USING EQUALITY CONDITIONS

Equality/inequality conditions are commonly used when modifying data. For example, let’s say that the movie rental company has a policy of removing old account rows once per year. Your task is to remove rows from the `rental` table where the rental date was in 2004. Here’s one way to tackle it:

```
DELETE FROM rental
WHERE year(rental_date) = 2004;
```

This statement includes a single equality condition; here’s an example which uses two inequality conditions to remove any rows where the rental date was not in 2005 or 2006:

```
DELETE FROM rental
WHERE year(rental_date) <> 2005 AND year(rental_date) <>
2006;
```

NOTE

When crafting examples of `delete` and `update` statements, I try to write each statement such that no rows are modified. That way, when you execute the statements, your data will remain unchanged, and your output from `select` statements will always match that shown in this book.

Since MySQL sessions are in auto-commit mode by default (see <<transactions>>), you would not be able to roll back (undo) any changes made to the example data if one of my statements modified the data. You may, of course, do whatever you want with the example data, including wiping it clean and rerunning the scripts to populate the tables, but I try to leave it intact.

Range Conditions

Along with checking that an expression is equal to (or not equal to) another expression, you can build conditions that check whether an expression falls within a certain range. This type of condition is common when working with numeric or temporal data. Consider the following query:

```
mysql> SELECT customer_id, rental_date
    -> FROM rental
    -> WHERE rental_date < '2005-05-25';
+-------------+---------------------+
| customer_id | rental_date         |
+-------------+---------------------+
|         130 | 2005-05-24 22:53:30 |
|         459 | 2005-05-24 22:54:33 |
|         408 | 2005-05-24 23:03:39 |
|         333 | 2005-05-24 23:04:41 |
|         222 | 2005-05-24 23:05:21 |
|         549 | 2005-05-24 23:08:07 |
|         269 | 2005-05-24 23:11:53 |
|         239 | 2005-05-24 23:31:46 |
+-------------+---------------------+
8 rows in set (0.00 sec)
```

This query finds all film rentals prior to May 25 2005. Along with specifying an upper limit for the rental date, you may also want to specify a lower range as well:

```
mysql> SELECT customer_id, rental_date
    -> FROM rental
    -> WHERE rental_date <= '2005-06-16'
    ->   AND rental_date >= '2005-06-14';
+-------------+---------------------+
| customer_id | rental_date         |
+-------------+---------------------+
|         416 | 2005-06-14 22:53:33 |
|         516 | 2005-06-14 22:55:13 |
|         239 | 2005-06-14 23:00:34 |
|         285 | 2005-06-14 23:07:08 |
|         310 | 2005-06-14 23:09:38 |
|         592 | 2005-06-14 23:12:46 |
...
|         148 | 2005-06-15 23:20:26 |
|         237 | 2005-06-15 23:36:37 |
|         155 | 2005-06-15 23:55:27 |
|         341 | 2005-06-15 23:57:20 |
|         149 | 2005-06-15 23:58:53 |
+-------------+---------------------+
364 rows in set (0.00 sec)
```

This version of the query retrieves all films rented on June 14 or 15 of 2005.

THE BETWEEN OPERATOR

When you have *both* an upper and lower limit for your range, you may choose to use a single condition that utilizes the `between` operator rather than using two separate conditions, as in:

```
mysql> SELECT customer_id, rental_date
    -> FROM rental
    -> WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-
16';
+-------------+---------------------+
```

```
| customer_id | rental_date         |
+-------------+---------------------+
|         416 | 2005-06-14 22:53:33 |
|         516 | 2005-06-14 22:55:13 |
|         239 | 2005-06-14 23:00:34 |
|         285 | 2005-06-14 23:07:08 |
|         310 | 2005-06-14 23:09:38 |
|         592 | 2005-06-14 23:12:46 |
…
|         148 | 2005-06-15 23:20:26 |
|         237 | 2005-06-15 23:36:37 |
|         155 | 2005-06-15 23:55:27 |
|         341 | 2005-06-15 23:57:20 |
|         149 | 2005-06-15 23:58:53 |
+-------------+---------------------+
364 rows in set (0.00 sec)
```

When using the `between` operator, there are a couple of things to keep in mind. You should always specify the lower limit of the range first (after `between`) and the upper limit of the range second (after `and`). Here's what happens if you mistakenly specify the upper limit first:

```
mysql> SELECT customer_id, rental_date
    -> FROM rental
    -> WHERE rental_date BETWEEN '2005-06-16' AND '2005-06-
14';
Empty set (0.00 sec)
```

As you can see, no data is returned. This is because the server is, in effect, generating two conditions from your single condition using the `<=` and `>=` operators, as in:

```
SELECT customer_id, rental_date
    -> FROM rental
    -> WHERE rental_date >= '2005-06-16'
    ->   AND rental_date <= '2005-06-14'
Empty set (0.00 sec)
```

Since it is impossible to have a date that is *both* greater than June 16 2005 and less than June 14 2005, the query returns an empty set. This brings me to the second pitfall when using `between`, which is to remember that your upper and lower limits are *inclusive*, meaning that the values you provide are included in the range limits. In this case, I want to return any films rented on the 14th or 15th of June, so I specify `2005-06-14` as the lower end of the range and `2005-06-16`as the upper end. Since I am not specifying the time component of the date, the time defaults to midnight, so the effective range is 2005-06-14 00:00:00 to 2005-06-16 00:00:00, which will include any rentals made on the 14th or 15th.

Along with dates, you can also build conditions to specify ranges of numbers. Numeric ranges are fairly easy to grasp, as demonstrated by the following:

```
mysql> SELECT customer_id, payment_date, amount
    -> FROM payment
    -> WHERE amount BETWEEN 10.0 AND 11.99;
+-------------+---------------------+--------+
| customer_id | payment_date        | amount |
+-------------+---------------------+--------+
|           2 | 2005-07-30 13:47:43 |  10.99 |
|           3 | 2005-07-27 20:23:12 |  10.99 |
|          12 | 2005-08-01 06:50:26 |  10.99 |
|          13 | 2005-07-29 22:37:41 |  11.99 |
|          21 | 2005-06-21 01:04:35 |  10.99 |
|          29 | 2005-07-09 21:55:19 |  10.99 |
...
|         571 | 2005-06-20 08:15:27 |  10.99 |
|         572 | 2005-06-17 04:05:12 |  10.99 |
|         573 | 2005-07-31 12:14:19 |  10.99 |
|         591 | 2005-07-07 20:45:51 |  11.99 |
|         592 | 2005-07-06 22:58:31 |  11.99 |
|         595 | 2005-07-31 11:51:46 |  10.99 |
+-------------+---------------------+--------+
114 rows in set (0.01 sec)
```

All payments between $10 and $11.99 are returned. Again, make sure that you specify the lower amount first.

STRING RANGES

While ranges of dates and numbers are easy to understand, you can also build conditions that search for ranges of strings, which are a bit harder to visualize. Say, for example, you are searching for customers whose last name falls within a range. Here's a query which returns customers whose last name falls between FA and FR:

```
mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE last_name BETWEEN 'FA' AND 'FR';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| FARNSWORTH | JOHN       |
| FENNELL    | ALEXANDER  |
| FERGUSON   | BERTHA     |
| FERNANDEZ  | MELINDA    |
| FIELDS     | VICKI      |
| FISHER     | CINDY      |
| FLEMING    | MYRTLE     |
| FLETCHER   | MAE        |
| FLORES     | JULIA      |
| FORD       | CRYSTAL    |
| FORMAN     | MICHEAL    |
| FORSYTHE   | ENRIQUE    |
| FORTIER    | RAUL       |
| FORTNER    | HOWARD     |
| FOSTER     | PHYLLIS    |
| FOUST      | JACK       |
| FOWLER     | JO         |
| FOX        | HOLLY      |
+------------+------------+
18 rows in set (0.00 sec)
```

While there are 5 customers whose last name starts with FR, they are not included in the results, since a name like FRANKLIN is outside of the range. However, we can pick up 4 of the 5 customers by extending the right-hand range to be FRB:

```
mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE last_name BETWEEN 'FA' AND 'FRB';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| FARNSWORTH | JOHN       |
| FENNELL    | ALEXANDER  |
| FERGUSON   | BERTHA     |
| FERNANDEZ  | MELINDA    |
| FIELDS     | VICKI      |
| FISHER     | CINDY      |
| FLEMING    | MYRTLE     |
| FLETCHER   | MAE        |
| FLORES     | JULIA      |
| FORD       | CRYSTAL    |
| FORMAN     | MICHEAL    |
| FORSYTHE   | ENRIQUE    |
| FORTIER    | RAUL       |
| FORTNER    | HOWARD     |
| FOSTER     | PHYLLIS    |
| FOUST      | JACK       |
| FOWLER     | JO         |
| FOX        | HOLLY      |
| FRALEY     | JUAN       |
| FRANCISCO  | JOEL       |
| FRANKLIN   | BETH       |
| FRAZIER    | GLENDA     |
+------------+------------+
22 rows in set (0.00 sec)
```

To work with string ranges, you need to know the order of the characters within your character set (the order in which the characters within a character set are sorted is called a *collation*).

Membership Conditions

In some cases, you will not be restricting an expression to a single value or range of values, but rather to a finite set of values. For example, you might want to locate all films which have a rating of either `'G'` or `'PG'`:

```
mysql> SELECT title, rating
    -> FROM film
    -> WHERE rating = 'G' OR rating = 'PG';
+---------------------------+--------+
| title                     | rating |
+---------------------------+--------+
| ACADEMY DINOSAUR          | PG     |
| ACE GOLDFINGER            | G      |
| AFFAIR PREJUDICE          | G      |
| AFRICAN EGG               | G      |
| AGENT TRUMAN              | PG     |
| ALAMO VIDEOTAPE           | G      |
| ALASKA PHANTOM            | PG     |
| ALI FOREVER               | PG     |
| AMADEUS HOLY              | PG     |
...
| WEDDING APOLLO            | PG     |
| WEREWOLF LOLA             | G      |
| WEST LION                 | G      |
| WIZARD COLDBLOODED        | PG     |
| WON DARES                 | PG     |
| WONDERLAND CHRISTMAS      | PG     |
| WORDS HUNTER              | PG     |
| WORST BANGER              | PG     |
| YOUNG LANGUAGE            | G      |
+---------------------------+--------+
372 rows in set (0.00 sec)
```

While this `where` clause (two conditions `or`‘d together) wasn’t too tedious to generate, imagine if the set of expressions contained 10 or 20 members. For these situations, you can use the `in` operator instead:

```
SELECT title, rating
FROM film
WHERE rating IN ('G','PG');
```

With the `in` operator, you can write a single condition no matter how many expressions are in the set.

USING SUBQUERIES

Along with writing your own set of expressions, such as `('G','PG')`, you can use a subquery to generate a set for you on the fly. For example, if you can assume that any film whose title includes the string `'PET'` would be safe for family viewing, you could execute a subquery against the `film` table to retrieve all ratings associated with these films, and then retrieve all films having any of these ratings:

```
mysql> SELECT title, rating
    -> FROM film
    -> WHERE rating IN (SELECT rating FROM film WHERE title
LIKE '%PET%');
+---------------------------+--------+
| title                     | rating |
+---------------------------+--------+
| ACADEMY DINOSAUR          | PG     |
| ACE GOLDFINGER            | G      |
| AFFAIR PREJUDICE          | G      |
| AFRICAN EGG               | G      |
| AGENT TRUMAN              | PG     |
| ALAMO VIDEOTAPE           | G      |
| ALASKA PHANTOM            | PG     |
| ALI FOREVER               | PG     |
| AMADEUS HOLY              | PG     |
...
| WEDDING APOLLO            | PG     |
| WEREWOLF LOLA             | G      |
| WEST LION                 | G      |
| WIZARD COLDBLOODED        | PG     |
| WON DARES                 | PG     |
| WONDERLAND CHRISTMAS      | PG     |
| WORDS HUNTER              | PG     |
| WORST BANGER              | PG     |
| YOUNG LANGUAGE            | G      |
```

```
+-------------------------+--------+
372 rows in set (0.00 sec)
```

The subquery returns the set `'G'` and `'PG'`, and the main query checks to see whether the value of the `rating` column can be found in the set returned by the subquery.

USING NOT IN

Sometimes you want to see whether a particular expression exists within a set of expressions, and sometimes you want to see whether the expression does *not* exist within the set. For these situations, you can use the `not in` operator:

```
SELECT title, rating
FROM film
WHERE rating NOT IN ('PG-13','R', 'NC-17');
```

This query finds all accounts that are *not* rated `'PG-13'` ,`'R'`, or `'NC-17'`, which will return the same set of 372 rows as the previous queries.

Matching Conditions

So far, you have been introduced to conditions that identify an exact string, a range of strings, or a set of strings; the final condition type deals with partial string matches. You may, for example, want to find all customers whose last name begins with *Q*. You could use a built-in function to strip off the first letter of the `last_name` column, as in:

```
mysql> SELECT last_name, first_name
    -> FROM customer
```

```
    -> WHERE left(last_name, 1) = 'Q';
+-------------+------------+
| last_name   | first_name |
+-------------+------------+
| QUALLS      | STEPHEN    |
| QUINTANILLA | ROGER      |
| QUIGLEY     | TROY       |
+-------------+------------+
3 rows in set (0.00 sec)
```

While the built-in function `left()` does the job, it doesn't give you much flexibility. Instead, you can use wildcard characters to build search expressions, as demonstrated in the next section.

USING WILDCARDS

When searching for partial string matches, you might be interested in:

- Strings beginning/ending with a certain character
- Strings beginning/ending with a substring
- Strings containing a certain character anywhere within the string
- Strings containing a substring anywhere within the string
- Strings with a specific format, regardless of individual characters

You can build search expressions to identify these and many other partial string matches by using the wildcard characters shown in Table 4-4.

Table 4-4. Wildcard characters

Wildcard character	Matches
`_`	Exactly one character
`%`	Any number of characters (including 0)

The underscore character takes the place of a single character, while the percent sign can take the place of a variable number of characters. When building conditions that utilize search expressions, you use the `like` operator, as in:

```
mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE last_name LIKE '_A_T%S';
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| MATTHEWS  | ERICA      |
| WALTERS   | CASSANDRA  |
| WATTS     | SHELLY     |
+-----------+------------+
3 rows in set (0.00 sec)
```

The search expression in the previous example specifies strings containing an *A* in the second position and a *T* in the fourth position, followed by any number of characters and ending in *S*. Table 4-5 shows some more search expressions and their interpretations.

Table 4-5. Sample search expressions

Search expression	Interpretation
`F%`	Strings beginning with *F*
`%t`	Strings ending with *t*
`%bas%`	Strings containing the substring `'bas'`
`_ _t_`	Four-character strings with a *t* in the third position
`_ _ _-_ _-_ _ _ _`	11-character strings with dashes in the fourth and seventh positions

The wildcard characters work fine for building simple search expressions; if your needs are a bit more sophisticated, however, you can use multiple search expressions, as demonstrated by the following:

```
mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE last_name LIKE 'Q%' OR last_name LIKE 'Y%';
+-------------+------------+
| last_name   | first_name |
+-------------+------------+
| QUALLS      | STEPHEN    |
| QUIGLEY     | TROY       |
| QUINTANILLA | ROGER      |
| YANEZ       | LUIS       |
| YEE         | MARVIN     |
| YOUNG       | CYNTHIA    |
+-------------+------------+
6 rows in set (0.00 sec)
```

This query finds all customers whose last name begins with *Q* or *Y*.

USING REGULAR EXPRESSIONS

If you find that the wildcard characters don't provide enough flexibility, you can use regular expressions to build search expressions. A regular expression is, in essence, a search expression on steroids. If you are new to SQL but have coded using programming languages such as Perl, then you might already be intimately familiar with regular expressions. If you have never used regular expressions, then you may want to consult Jeffrey E.F. Friedl's *Mastering Regular Expressions* (O'Reilly), since it is far too large a topic to try to cover in this book.

Here's what the previous query (find all customers whose last name starts with *Q* or *Y*) would look like using the MySQL implementation of regular expressions:

```
mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE last_name REGEXP '^[QY]';
+-------------+------------+
```

```
| last_name   | first_name |
+-------------+------------+
| YOUNG       | CYNTHIA    |
| QUALLS      | STEPHEN    |
| QUINTANILLA | ROGER      |
| YANEZ       | LUIS       |
| YEE         | MARVIN     |
| QUIGLEY     | TROY       |
+-------------+------------+
6 rows in set (0.16 sec)
```

The `regexp` operator takes a regular expression (`'^[QY]'` in this example) and applies it to the expression on the left-hand side of the condition (the column `last_name`). The query now contains a single condition using a regular expression rather than two conditions using wildcard characters.

Oracle Database and Microsoft SQL Server also support regular expressions. With Oracle Database, you would use the `regexp_like` function instead of the `regexp` operator shown in the previous example, whereas SQL Server allows regular expressions to be used with the `like` operator.

Null: That Four-Letter Word

I put it off as long as I could, but it's time to broach a topic that tends to be met with fear, uncertainty, and dread: the `null` value. `Null` is the absence of a value; before an employee is terminated, for example, her `end_date` column in the `employee` table should be `null`. There is no value that can be assigned to the `end_date` column that would make sense in this situation. `Null` is a bit slippery, however, as there are various flavors of `null`:

Not applicable

Such as the employee ID column for a transaction that took place at an ATM machine

Value not yet known

Such as when the federal ID is not known at the time a customer row is created

Value undefined

Such as when an account is created for a product that has not yet been added to the database

> **NOTE**
>
> Some theorists argue that there should be a different expression to cover each of these (and more) situations, but most practitioners would agree that having multiple `null` values would be far too confusing.

When working with `null`, you should remember:

- An expression can *be* null, but it can never *equal* null.
- Two nulls are never equal to each other.

To test whether an expression is `null`, you need to use the `is null` operator, as demonstrated by the following:

```
mysql> SELECT rental_id, customer_id
    -> FROM rental
    -> WHERE return_date IS NULL;
+-----------+-------------+
| rental_id | customer_id |
+-----------+-------------+
|     11496 |         155 |
```

```
|     11541 |         335 |
|     11563 |          83 |
|     11577 |         219 |
|     11593 |          99 |
...
|     15867 |         505 |
|     15875 |          41 |
|     15894 |         168 |
|     15966 |         374 |
+-----------+-------------+
183 rows in set (0.01 sec)
```

This query finds all film rentals which were never returned. Here’s the same query using `= null` instead of `is null`:

```
mysql> SELECT rental_id, customer_id
    -> FROM rental
    -> WHERE return_date = NULL;
Empty set (0.01 sec)
```

As you can see, the query parses and executes but does not return any rows. This is a common mistake made by inexperienced SQL programmers, and the database server will not alert you to your error, so be careful when constructing conditions that test for `null`.

If you want to see whether a value has been assigned to a column, you can use the `is not null` operator, as in:

```
mysql> SELECT rental_id, customer_id, return_date
    -> FROM rental
    -> WHERE return_date IS NOT NULL
    -> limit 20;
+-----------+-------------+---------------------+
| rental_id | customer_id | return_date         |
+-----------+-------------+---------------------+
|         1 |         130 | 2005-05-26 22:04:30 |
|         2 |         459 | 2005-05-28 19:40:33 |
|         3 |         408 | 2005-06-01 22:12:39 |
|         4 |         333 | 2005-06-03 01:43:41 |
|         5 |         222 | 2005-06-02 04:33:21 |
```

```
|         6 |         549 | 2005-05-27 01:32:07 |
|         7 |         269 | 2005-05-29 20:34:53 |
...
|     16043 |         526 | 2005-08-31 03:09:03 |
|     16044 |         468 | 2005-08-25 04:08:39 |
|     16045 |          14 | 2005-08-25 23:54:26 |
|     16046 |          74 | 2005-08-27 18:02:47 |
|     16047 |         114 | 2005-08-25 02:48:48 |
|     16048 |         103 | 2005-08-31 21:33:07 |
|     16049 |         393 | 2005-08-30 01:01:12 |
+-----------+-------------+---------------------+
15861 rows in set (0.02 sec)
```

This version of the query returns all rentals which were returned, which is the majority of the rows in the table (15,861 out of 16,044).

Before putting `null` aside for a while, it would be helpful to investigate one more potential pitfall. Suppose that you have been asked to find all rentals which were not returned during May through August of 2005. Your first instinct might be to do the following:

```
mysql> SELECT rental_id, customer_id, return_date
    -> FROM rental
    -> WHERE return_date NOT BETWEEN '2005-05-01' AND '2005-
09-01';
+-----------+-------------+---------------------+
| rental_id | customer_id | return_date         |
+-----------+-------------+---------------------+
|     15365 |         327 | 2005-09-01 03:14:17 |
|     15388 |          50 | 2005-09-01 03:50:23 |
|     15392 |         410 | 2005-09-01 01:14:15 |
|     15401 |         103 | 2005-09-01 03:44:10 |
|     15415 |         204 | 2005-09-01 02:05:56 |
...
|     15977 |         550 | 2005-09-01 22:12:10 |
|     15982 |         370 | 2005-09-01 21:51:31 |
|     16005 |         466 | 2005-09-02 02:35:22 |
|     16020 |         311 | 2005-09-01 18:17:33 |
|     16033 |         226 | 2005-09-01 02:36:15 |
|     16037 |          45 | 2005-09-01 02:48:04 |
|     16040 |         195 | 2005-09-02 02:19:33 |
```

```
+-----------+------------+--------------------+
62 rows in set (0.01 sec)
```

While it is true that these 62 rentals were returned outside of the May to August window, if you look carefully at the data, you will see that all of the rows returned have a non-Null return date. But what about the 183 rentals which were never returned? One might argue that these 183 rows were also not returned between May and August, so they should also be included in the result set. To answer the question correctly, therefore, you need to account for the possibility that some rows might contain a `null` in the `return_date` column:

```
mysql> SELECT rental_id, customer_id, return_date
    -> FROM rental
    -> WHERE return_date IS NULL
    ->   OR return_date NOT BETWEEN '2005-05-01' AND '2005-
09-01';
+-----------+------------+---------------------+
| rental_id | customer_id | return_date         |
+-----------+------------+---------------------+
|     11496 |         155 | NULL                |
|     11541 |         335 | NULL                |
|     11563 |          83 | NULL                |
|     11577 |         219 | NULL                |
|     11593 |          99 | NULL                |
...
|     15939 |         382 | 2005-09-01 17:25:21 |
|     15942 |         210 | 2005-09-01 18:39:40 |
|     15966 |         374 | NULL                |
|     15971 |         187 | 2005-09-02 01:28:33 |
|     15973 |         343 | 2005-09-01 20:08:41 |
|     15977 |         550 | 2005-09-01 22:12:10 |
|     15982 |         370 | 2005-09-01 21:51:31 |
|     16005 |         466 | 2005-09-02 02:35:22 |
|     16020 |         311 | 2005-09-01 18:17:33 |
|     16033 |         226 | 2005-09-01 02:36:15 |
|     16037 |          45 | 2005-09-01 02:48:04 |
|     16040 |         195 | 2005-09-02 02:19:33 |
+-----------+------------+---------------------+
245 rows in set (0.01 sec)
```

The result set now includes the 62 rentals which were returned outside of the May to August window, along with the 183 rentals which were never returned, for a total of 245 rows. When working with a database that you are not familiar with, it is a good idea to find out which columns in a table allow `null`s so that you can take appropriate measures with your filter conditions to keep data from slipping through the cracks.

Test Your Knowledge

The following exercises test your understanding of filter conditions. Please see Appendix C for solutions.

The following subset of rows from the Payment table are used for the first two exercises:

```
+------------+-------------+--------+--------------------+
| payment_id | customer_id | amount | date(payment_date) |
+------------+-------------+--------+--------------------+
|        101 |           4 |   8.99 | 2005-08-18         |
|        102 |           4 |   1.99 | 2005-08-19         |
|        103 |           4 |   2.99 | 2005-08-20         |
|        104 |           4 |   6.99 | 2005-08-20         |
|        105 |           4 |   4.99 | 2005-08-21         |
|        106 |           4 |   2.99 | 2005-08-22         |
|        107 |           4 |   1.99 | 2005-08-23         |
|        108 |           5 |   0.99 | 2005-05-29         |
|        109 |           5 |   6.99 | 2005-05-31         |
|        110 |           5 |   1.99 | 2005-05-31         |
|        111 |           5 |   3.99 | 2005-06-15         |
|        112 |           5 |   2.99 | 2005-06-16         |
|        113 |           5 |   4.99 | 2005-06-17         |
|        114 |           5 |   2.99 | 2005-06-19         |
|        115 |           5 |   4.99 | 2005-06-20         |
|        116 |           5 |   4.99 | 2005-07-06         |
|        117 |           5 |   2.99 | 2005-07-08         |
|        118 |           5 |   4.99 | 2005-07-09         |
```

```
|        119 |           5 |   5.99 | 2005-07-09         |
|        120 |           5 |   1.99 | 2005-07-09         |
+------------+-------------+--------+--------------------+
```

Exercise 4-1

Which of the payment IDs would be returned by the following filter conditions?

```
customer_id <> 5 AND (amount > 8 OR date(payment_date) =
'2005-08-23')
```

Exercise 4-2

Which of the payment IDs would be returned by the following filter conditions?

```
customer_id = 5 AND NOT (amount > 6 OR date(payment_date) =
'2005-06-19')
```

Exercise 4-3

Construct a query that retrieves all rows from the Payment table where the amount is either 1.98, 7.98, or 9.98.

Exercise 4-4

Construct a query that finds all customers whose last name contains an *A* in the second position and a *W* anywhere after the *A*.

Chapter 5. Querying Multiple Tables

Back in Chapter 2, I demonstrated how related concepts are broken into separate pieces through a process known as normalization. The end result of this exercise was two tables: `person` and `favorite_food`. If, however, you want to generate a single report showing a person's name, address, *and* favorite foods, you will need a mechanism to bring the data from these two tables back together again; this mechanism is known as a *join*, and this chapter concentrates on the simplest and most common join, the *inner join*. Chapter 10 demonstrates all of the different join types.

What Is a Join?

Queries against a single table are certainly not rare, but you will find that most of your queries will require two, three, or even more tables. To illustrate, let's look at the definitions for the `customer` and `address` tables and then define a query that retrieves data from both tables:

```
mysql> desc customer;
+-------------+----------------------+------+-----+--------
----------+
| Field       | Type                 | Null | Key |
Default          |
+-------------+----------------------+------+-----+--------
----------+
```

```
| customer_id | smallint(5) unsigned | NO   | PRI |
NULL              |
| store_id    | tinyint(3) unsigned  | NO   | MUL |
NULL              |
| first_name  | varchar(45)          | NO   |     |
NULL              |
| last_name   | varchar(45)          | NO   | MUL |
NULL              |
| email       | varchar(50)          | YES  |     |
NULL              |
| address_id  | smallint(5) unsigned | NO   | MUL |
NULL              |
| active      | tinyint(1)           | NO   |     |
1                 |
| create_date | datetime             | NO   |     |
NULL              |
| last_update | timestamp            | YES  |     |
CURRENT_TIMESTAMP |
+-------------+----------------------+------+-----+---------
----------+

mysql> desc address;
+-------------+----------------------+------+-----+---------
----------+
| Field       | Type                 | Null | Key |
Default           |
+-------------+----------------------+------+-----+---------
----------+
| address_id  | smallint(5) unsigned | NO   | PRI |
NULL              |
| address     | varchar(50)          | NO   |     |
NULL              |
| address2    | varchar(50)          | YES  |     |
NULL              |
| district    | varchar(20)          | NO   |     |
NULL              |
| city_id     | smallint(5) unsigned | NO   | MUL |
NULL              |
| postal_code | varchar(10)          | YES  |     |
NULL              |
| phone       | varchar(20)          | NO   |     |
NULL              |
| location    | geometry             | NO   | MUL |
NULL              |
| last_update | timestamp            | NO   |     |
CURRENT_TIMESTAMP |
+-------------+----------------------+------+-----+---------
----------+
```

Let’s say you want to retrieve the first and last names of each customer, along with their street address. Your query will therefore need to retrieve the `customer.first_name`, `customer.last_name`, and `address.address` columns. But how can you retrieve data from both tables in the same query? The answer lies in the `customer.address_id` column, which holds the ID of the customer’s record in the `address` table (in more formal terms, the `customer.address_id` column is the *foreign key* to the `address` table). The query, which you will see shortly, instructs the server to use the `customer.address_id` column as the *transportation* between the `customer` and `address` tables, thereby allowing columns from both tables to be included in the query’s result set. This type of operation is known as a join.

NOTE

A foreign key constraint can optionally be created to verify that the values in one table exist in another table. For the previous example, a foreign key constraint could be created on the `customer` table to ensure that any values inserted into the `customer.address_id` column can be found in the `address.address_id` column. Please note that it is not necessary to have a foreign key constraint in place in order to join two tables.

Cartesian Product

The easiest way to start is to put the `customer` and `address` tables into the `from` clause of a query and see what happens. Here’s a query that retrieves the customer’s first and last names along with the street address, with a `from` clause naming both tables separated by the `join` keyword:

```
mysql> SELECT c.first_name, c.last_name, a.address
    -> FROM customer c JOIN address;
+------------+-----------+---------------------+
| first_name | last_name | address             |
+------------+-----------+---------------------+
| MARY       | SMITH     | 47 MySakila Drive   |
| PATRICIA   | JOHNSON   | 47 MySakila Drive   |
| LINDA      | WILLIAMS  | 47 MySakila Drive   |
| BARBARA    | JONES     | 47 MySakila Drive   |
| ELIZABETH  | BROWN     | 47 MySakila Drive   |
| JENNIFER   | DAVIS     | 47 MySakila Drive   |
| MARIA      | MILLER    | 47 MySakila Drive   |
| SUSAN      | WILSON    | 47 MySakila Drive   |
...
| SETH       | HANNON    | 1325 Fukuyama Street |
| KENT       | ARSENAULT | 1325 Fukuyama Street |
| TERRANCE   | ROUSH     | 1325 Fukuyama Street |
| RENE       | MCALISTER | 1325 Fukuyama Street |
| EDUARDO    | HIATT     | 1325 Fukuyama Street |
| TERRENCE   | GUNDERSON | 1325 Fukuyama Street |
| ENRIQUE    | FORSYTHE  | 1325 Fukuyama Street |
| FREDDIE    | DUGGAN    | 1325 Fukuyama Street |
| WADE       | DELVALLE  | 1325 Fukuyama Street |
| AUSTIN     | CINTRON   | 1325 Fukuyama Street |
+------------+-----------+---------------------+
361197 rows in set (0.03 sec)
```

Hmmm…there are only 599 customers and 603 rows in the `address` table, so how did the result set end up with 361,197 rows? Looking more closely, you can see that many of the customers seem to have the same street address. Because the query didn't specify *how* the two tables should be joined, the database server generated the *Cartesian product*, which is *every* permutation of the two tables (599 customers x 603 addresses = 361,197 permutations). This type of join is known as a *cross join*, and it is rarely used (on purpose, at least). Cross joins are one of the join types that we study in Chapter 10.

Inner Joins

To modify the previous query so that only a single row is returned for each customer, you need to describe how the two tables are related. Earlier, I showed that the `customer.address_id` column serves as the link between the two tables, so this information needs to be added to the `on` subclause of the `from` clause:

```
mysql> SELECT c.first_name, c.last_name, a.address
    -> FROM customer c JOIN address a
    ->   ON c.address_id = a.address_id;
+------------+------------+------------------------------
----------+
| first_name | last_name  |
address                                |
+------------+------------+------------------------------
----------+
| MARY       | SMITH      | 1913 Hanoi
Way                           |
| PATRICIA   | JOHNSON    | 1121 Loja
Avenue                          |
| LINDA      | WILLIAMS   | 692 Joliet
Street                        |
| BARBARA    | JONES      | 1566 Inegl
Manor                         |
| ELIZABETH  | BROWN      | 53 Idfu
Parkway                          |
| JENNIFER   | DAVIS      | 1795 Santiago de Compostela
Way        |
| MARIA      | MILLER     | 900 Santiago de Compostela
Parkway     |
| SUSAN      | WILSON     | 478 Joliet
Way                           |
| MARGARET   | MOORE      | 613 Korolev
Drive                       |
...
| TERRANCE   | ROUSH      | 42 Fontana
Avenue                        |
| RENE       | MCALISTER  | 1895 Zhezqazghan
Drive                 |
| EDUARDO    | HIATT      | 1837 Kaduna
Parkway                     |
| TERRENCE   | GUNDERSON  | 844 Bucuresti
```

```
Place                    |
| ENRIQUE     | FORSYTHE     | 1101 Bucuresti
Boulevard               |
| FREDDIE     | DUGGAN       | 1103 Quilmes
Boulevard                 |
| WADE        | DELVALLE     | 1331 Usak
Boulevard                    |
| AUSTIN      | CINTRON      | 1325 Fukuyama
Street                   |
+-------------+-------------+------------------------------
----------+
599 rows in set (0.00 sec)
```

Instead of 361,197 rows, you now have the expected 599 rows due to the addition of the `on` subclause, which instructs the server to join the `customer` and `address` tables by using the `address_id` column to traverse from one table to the other. For example, Mary Smith's row in the `customer` table contains a value of `5` in the `address_id` column (not shown in the example). The server uses this value to look up the row in the `address` table having a value of `5` in its `address_id` column and then retrieves the value `'1913 Hanoi Way'` from the `address` column in that row.

If a value exists for the `address_id` column in one table but *not* the other, then the join fails for the rows containing that value and those rows are excluded from the result set. This type of join is known as an *inner join*, and it is the most commonly used type of join. To clarify, if a row in the `customer` table has the value 999 in the `address_id` column, and there's no row in the `address` table with a value of 999 in the `address_id` column, then that customer row would not be included in the result set. If you want to include all rows from one table or the other regardless of whether a match exists, you need to specify an *outer join*, but we cover this later in the book.

In the previous example, I did not specify in the `from` clause which type of join to use. However, when you wish to join two tables using an inner join, you should explicitly specify this in your `from` clause; here's the same example, with the addition of the join type (note the keyword `INNER`):

```
SELECT c.first_name, c.last_name, a.address
FROM customer c INNER JOIN address a
  ON c.address_id = a.address_id;
```

If you do not specify the type of join, then the server will do an inner join by default. As you will see later in the book, however, there are several types of joins, so you should get in the habit of specifying the exact type of join that you require, especially for the benefit of any other people who might use/maintain your queries in the future.

If the names of the columns used to join the two tables are identical, which is true in the previous query, you can use the `using` subclause instead of the `on` subclause, as in:

```
SELECT c.first_name, c.last_name, a.address
FROM customer c INNER JOIN address a
  USING (address_id);
```

Since `using` is a shorthand notation that you can use in only a specific situation, I prefer always to use the `on` subclause to avoid confusion.

The ANSI Join Syntax

The notation used throughout this book for joining tables was introduced in the SQL92 version of the ANSI SQL standard. All the major databases (Oracle Database, Microsoft SQL Server, MySQL, IBM DB2 Universal Database, and Sybase Adaptive Server) have adopted the SQL92 join syntax. Because most of these servers have been around since before the release of the SQL92 specification, they all include an older join syntax as well. For example, all these servers would understand the following variation of the previous query:

```
mysql> SELECT c.first_name, c.last_name, a.address
    -> FROM customer c, address a
    -> WHERE c.address_id = a.address_id;
+------------+-----------+------------------------------------
---+
| first_name | last_name  |
address                             |
+------------+-----------+------------------------------------
---+
| MARY       | SMITH      | 1913 Hanoi
Way                      |
| PATRICIA   | JOHNSON    | 1121 Loja
Avenue                    |
| LINDA      | WILLIAMS   | 692 Joliet
Street                   |
| BARBARA    | JONES      | 1566 Inegl
Manor                    |
| ELIZABETH  | BROWN      | 53 Idfu
Parkway                      |
| JENNIFER   | DAVIS      | 1795 Santiago de Compostela
Way    |
| MARIA      | MILLER     | 900 Santiago de Compostela
Parkway |
| SUSAN      | WILSON     | 478 Joliet
Way                      |
| MARGARET   | MOORE      | 613 Korolev
Drive                  |
...
| TERRANCE   | ROUSH      | 42 Fontana Avenue
|
| RENE       | MCALISTER  | 1895 Zhezqazghan Drive
|
| EDUARDO    | HIATT      | 1837 Kaduna Parkway
 |
| TERRENCE   | GUNDERSON  | 844 Bucuresti Place
 |
```

```
| ENRIQUE    | FORSYTHE   | 1101 Bucuresti Boulevard
 |
| FREDDIE    | DUGGAN     | 1103 Quilmes Boulevard
 |
| WADE       | DELVALLE   | 1331 Usak Boulevard
 |
| AUSTIN     | CINTRON    | 1325 Fukuyama Street
  |
+------------+------------+------------------------------
---+
599 rows in set (0.00 sec)
```

This older method of specifying joins does not include the `on` subclause; instead, tables are named in the `from` clause separated by commas, and join conditions are included in the `where` clause. While you may decide to ignore the SQL92 syntax in favor of the older join syntax, the ANSI join syntax has the following advantages:

- Join conditions and filter conditions are separated into two different clauses (the `on` subclause and the `where` clause, respectively), making a query easier to understand.
- The join conditions for each pair of tables are contained in their own `on` clause, making it less likely that part of a join will be mistakenly omitted.
- Queries that use the SQL92 join syntax are portable across database servers, whereas the older syntax is slightly different across the different servers.

The benefits of the SQL92 join syntax are easier to identify for complex queries that include both join and filter conditions. Consider the following query, which returns only those customers whose postal code is 52137:

```
mysql> SELECT c.first_name, c.last_name, a.address
    -> FROM customer c, address a
```

```
    -> WHERE c.address_id = a.address_id
    ->   AND a.postal_code = 52137;
+------------+-----------+------------------------+
| first_name | last_name | address                |
+------------+-----------+------------------------+
| JAMES      | GANNON    | 1635 Kuwana Boulevard  |
| FREDDIE    | DUGGAN    | 1103 Quilmes Boulevard |
+------------+-----------+------------------------+
2 rows in set (0.01 sec)
```

At first glance, it is not so easy to determine which conditions in the `where` clause are join conditions and which are filter conditions. It is also not readily apparent which type of join is being employed (to identify the type of join, you would need to look closely at the join conditions in the `where` clause to see whether any special characters are employed), nor is it easy to determine whether any join conditions have been mistakenly left out. Here's the same query using the SQL92 join syntax:

```
mysql> SELECT c.first_name, c.last_name, a.address
    -> FROM customer c INNER JOIN address a
    ->   ON c.address_id = a.address_id
    -> WHERE a.postal_code = 52137;
+------------+-----------+------------------------+
| first_name | last_name | address                |
+------------+-----------+------------------------+
| JAMES      | GANNON    | 1635 Kuwana Boulevard  |
| FREDDIE    | DUGGAN    | 1103 Quilmes Boulevard |
+------------+-----------+------------------------+
2 rows in set (0.00 sec)
```

With this version, it is clear which condition is used for the join, and which condition is used for filtering. Hopefully, you will agree that the version using SQL92 join syntax is easier to understand.

Joining Three or More Tables

Joining three tables is similar to joining two tables, but with one slight wrinkle. With a two-table join, there are two tables and one join type in the `from` clause, and a single `on` subclause to define how the tables are joined. With a three-table join, there are three tables and two join types in the `from` clause, and two `on` subclauses.

To illustrate, let's change the previous query to return the customer's city rather than their street address. The city name, however, is not stored in the `address` table, but is accessed via a foreign key to the `city` table. Here are the table definitions:

```
mysql> desc address;
+-------------+----------------------+------+-----+---------
----------+
| Field       | Type                 | Null | Key |
Default           |
+-------------+----------------------+------+-----+---------
----------+
| address_id  | smallint(5) unsigned | NO   | PRI |
NULL              |
| address     | varchar(50)          | NO   |     |
NULL              |
| address2    | varchar(50)          | YES  |     |
NULL              |
| district    | varchar(20)          | NO   |     |
NULL              |
| city_id     | smallint(5) unsigned | NO   | MUL |
NULL              |
| postal_code | varchar(10)          | YES  |     |
NULL              |
| phone       | varchar(20)          | NO   |     |
NULL              |
| location    | geometry             | NO   | MUL |
NULL              |
| last_update | timestamp            | NO   |     |
CURRENT_TIMESTAMP |
+-------------+----------------------+------+-----+---------
----------+

mysql> desc city;
```

```
+-------------+---------------------+------+-----+---------
----------+
| Field       | Type                | Null | Key |
Default           |
+-------------+---------------------+------+-----+---------
----------+
| city_id     | smallint(5) unsigned | NO   | PRI |
NULL              |
| city        | varchar(50)          | NO   |     |
NULL              |
| country_id  | smallint(5) unsigned | NO   | MUL |
NULL              |
| last_update | timestamp            | NO   |     |
CURRENT_TIMESTAMP |
+-------------+---------------------+------+-----+---------
----------+
```

In order to show each customer's city, you will need to traverse from the `customer` table to the `address` table using the `address_id` column, and then from the `address` table to the `city` table using the `city_id` column. Here's what the query would look like:

```
mysql> SELECT c.first_name, c.last_name, ct.city
    -> FROM customer c
    ->   INNER JOIN address a
    ->   ON c.address_id = a.address_id
    ->   INNER JOIN city ct
    ->   ON a.city_id = ct.city_id;
+-------------+-------------+----------------------------+
| first_name  | last_name   | city                       |
+-------------+-------------+----------------------------+
| JULIE       | SANCHEZ     | A Corua (La Corua)         |
| PEGGY       | MYERS       | Abha                       |
| TOM         | MILNER      | Abu Dhabi                  |
| GLEN        | TALBERT     | Acua                       |
| LARRY       | THRASHER    | Adana                      |
| SEAN        | DOUGLASS    | Addis Abeba                |
...
| MICHELE     | GRANT       | Yuncheng                   |
| GARY        | COY         | Yuzhou                     |
| PHYLLIS     | FOSTER      | Zalantun                   |
| CHARLENE    | ALVAREZ     | Zanzibar                   |
| FRANKLIN    | TROUTMAN    | Zaoyang                    |
| FLOYD       | GANDY       | Zapopan                    |
```

```
| CONSTANCE   | REID         | Zaria                       |
| JACK        | FOUST        | Zeleznogorsk                |
| BYRON       | BOX          | Zhezqazghan                 |
| GUY         | BROWNLEE     | Zhoushan                    |
| RONNIE      | RICKETTS     | Ziguinchor                  |
+-------------+--------------+-----------------------------+
599 rows in set (0.03 sec)
```

For this query, there are three tables, two join types, and two `on` subclauses in the `from` clause, so things have gotten quite a bit busier. At first glance, it might seem like the order in which the tables appear in the `from` clause is important, but if you switch the table order you will get the exact same results. All three of these variations return the same results:

```
SELECT c.first_name, c.last_name, ct.city
FROM customer c
  INNER JOIN address a
  ON c.address_id = a.address_id
  INNER JOIN city ct
  ON a.city_id = ct.city_id;

SELECT c.first_name, c.last_name, ct.city
FROM city ct
  INNER JOIN address a
  ON a.city_id = ct.city_id
  INNER JOIN customer c
  ON c.address_id = a.address_id;

SELECT c.first_name, c.last_name, ct.city
FROM address a
  INNER JOIN city ct
  ON a.city_id = ct.city_id
  INNER JOIN customer c
  ON c.address_id = a.address_id;
```

The only difference you may see would be the order in which the rows are returned, since there is no `order by` clause to specify how the results should be ordered.

DOES JOIN ORDER MATTER?

If you are confused about why all three versions of the `customer`/`address`/`city` query yield the same results, keep in mind that SQL is a nonprocedural language, meaning that you describe what you want to retrieve and which database objects need to be involved, but it is up to the database server to determine how best to execute your query. Using statistics gathered from your database objects, the server must pick one of three tables as a starting point (the chosen table is thereafter known as the *driving table*), and then decide in which order to join the remaining tables. Therefore, the order in which tables appear in your `from` clause is not significant.

If, however, you believe that the tables in your query should always be joined in a particular order, you can place the tables in the desired order and then specify the keyword `STRAIGHT_JOIN` in MySQL, request the `FORCE ORDER` option in SQL Server, or use either the `ORDERED` or the `LEADING` optimizer hint in Oracle Database. For example, to tell the MySQL server to use the `city` table as the driving table and to then join the `address` and `customer` tables, you could do the following:

```
SELECT STRAIGHT_JOIN c.first_name, c.last_name, ct.city
FROM city ct
  INNER JOIN address a
  ON a.city_id = ct.city_id
  INNER JOIN customer c
  ON c.address_id = a.address_id
```

Using Subqueries As Tables

You have already seen several examples of queries that include multiple tables, but there is one variation worth mentioning: what to do if some of the data sets are generated by subqueries. Subqueries are the focus of Chapter 9, but I already introduced the concept of a subquery in the `from` clause in the previous chapter. Here's a query which joins the `customer` table to a subquery against the `address` and `city` tables:

```
mysql> SELECT c.first_name, c.last_name, addr.address,
addr.city
    -> FROM customer c
    ->   INNER JOIN
    ->    (SELECT a.address_id, a.address, ct.city
    ->     FROM address a
    ->       INNER JOIN city ct
```

```
    ->       ON a.city_id = ct.city_id
    ->     WHERE a.district = 'California'
    ->    ) addr
    ->   ON c.address_id = addr.address_id;
+------------+-----------+------------------------+---------
-------+
| first_name | last_name | address                |
city           |
+------------+-----------+------------------------+---------
-------+
| PATRICIA   | JOHNSON   | 1121 Loja Avenue       | San
Bernardino |
| BETTY      | WHITE     | 770 Bydgoszcz Avenue   | Citrus
Heights |
| ALICE      | STEWART   | 1135 Izumisano Parkway |
Fontana        |
| ROSA       | REYNOLDS  | 793 Cam Ranh Avenue    |
Lancaster      |
| RENEE      | LANE      | 533 al-Ayn Boulevard   |
Compton        |
| KRISTIN    | JOHNSTON  | 226 Brest Manor        |
Sunnyvale      |
| CASSANDRA  | WALTERS   | 920 Kumbakonam Loop    |
Salinas        |
| JACOB      | LANCE     | 1866 al-Qatif Avenue   | El
Monte       |
| RENE       | MCALISTER | 1895 Zhezqazghan Drive | Garden
Grove   |
+------------+-----------+------------------------+---------
-------+
9 rows in set (0.00 sec)
```

The subquery, which starts on line 4 and is given the alias `addr`, finds all addresses which are in California. The outer query joins the subquery results to the `customer` table to return the first name, last name, street address, and city of all customers who live in California. While this query could have been written without the use of a subquery by simply joining the three tables, it can sometimes be advantageous from a performance and/or readability aspect to use one or more subqueries.

One way to visualize what is going on is to run the subquery by itself and look at the results. Here are the results of the subquery from the prior example:

```
mysql> SELECT a.address_id, a.address, ct.city
    -> FROM address a
    ->   INNER JOIN city ct
    ->   ON a.city_id = ct.city_id
    -> WHERE a.district = 'California';
+------------+------------------------+----------------+
| address_id | address                | city           |
+------------+------------------------+----------------+
|          6 | 1121 Loja Avenue       | San Bernardino |
|         18 | 770 Bydgoszcz Avenue   | Citrus Heights |
|         55 | 1135 Izumisano Parkway | Fontana        |
|        116 | 793 Cam Ranh Avenue    | Lancaster      |
|        186 | 533 al-Ayn Boulevard   | Compton        |
|        218 | 226 Brest Manor        | Sunnyvale      |
|        274 | 920 Kumbakonam Loop    | Salinas        |
|        425 | 1866 al-Qatif Avenue   | El Monte       |
|        599 | 1895 Zhezqazghan Drive | Garden Grove   |
+------------+------------------------+----------------+
9 rows in set (0.00 sec)
```

This result set consists of all 9 California addresses. When joined to the `customer` table via the `address_id` column, your result set will contain information about the customers assigned to these addresses.

Using the Same Table Twice

If you are joining multiple tables, you might find that you need to join the same table more than once. In the sample database, for example, actors are related to the films in which they appeared via the `film_actor` table. If you want to find all of the films in which two specific actors appear, you could write a query such as this one, which joins the `film` table to the `film_actor` table to the `actor` table:

```
mysql> SELECT f.title
    -> FROM film f
    ->   INNER JOIN film_actor fa
    ->   ON f.film_id = fa.film_id
    ->   INNER JOIN actor a
    ->   ON fa.actor_id = a.actor_id
    -> WHERE ((a.first_name = 'CATE' AND a.last_name =
'MCQUEEN')
    ->     OR (a.first_name = 'CUBA' AND a.last_name =
'BIRCH'));
+----------------------+
| title                |
+----------------------+
| ATLANTIS CAUSE       |
| BLOOD ARGONAUTS      |
| COMMANDMENTS EXPRESS |
| DYNAMITE TARZAN      |
| EDGE KISSING         |
...
| TOWERS HURRICANE     |
| TROJAN TOMORROW      |
| VIRGIN DAISY         |
| VOLCANO TEXAS        |
| WATERSHIP FRONTIER   |
+----------------------+
54 rows in set (0.00 sec)
```

This query returns all movies in which either Cate McQueen or Cuba Birch appeared. However, let's say that you want to retrieve only those films in which *both* of these actors appeared. To accomplish this, you will need to find all rows in the `film` table which have two rows in the `film_actor` table, one of which is associated with Cate McQueen, and the other one associated with Cuba Birch. Therefore, you will need to include the `film_actor` and `actor` tables twice, each with a different alias so that the server knows which one you are referring to in the various clauses:

```
mysql> SELECT f.title
    ->  FROM film f
    ->    INNER JOIN film_actor fa1
    ->    ON f.film_id = fa1.film_id
    ->    INNER JOIN actor a1
    ->    ON fa1.actor_id = a1.actor_id
    ->    INNER JOIN film_actor fa2
    ->    ON f.film_id = fa2.film_id
    ->    INNER JOIN actor a2
    ->    ON fa2.actor_id = a2.actor_id
    -> WHERE (a1.first_name = 'CATE' AND a1.last_name =
'MCQUEEN')
    ->   AND (a2.first_name = 'CUBA' AND a2.last_name =
'BIRCH');
+------------------+
| title            |
+------------------+
| BLOOD ARGONAUTS  |
| TOWERS HURRICANE |
+------------------+
2 rows in set (0.00 sec)
```

Between them, the two actors appeared in 52 different films, but there are only 2 films in which both actors appeared. This is one example of a query that *requires* the use of table aliases, since the same tables are used multiple times.

Self-Joins

Not only can you include the same table more than once in the same query, but you can actually join a table to itself. This might seem like a strange thing to do at first, but there are valid reasons for doing so. Some tables include a *self-referencing foreign key*, which means that it includes a column which points to the primary key within the same table. While the sample database doesn't include such a relationship, let's imagine that the `film` table includes the column `prequel_film_id`, which points to the film's parent (e.g. the film

“Fiddler Lost II” would use this column to point to the parent film “Fiddler Lost”). Here’s what the table would look like if we were to add this additional column:

```
mysql> desc film;
+----------------------+------------------------------------
------+------+-----+-------------------+
| Field                |
Type                                  | Null | Key |
Default            |
+----------------------+------------------------------------
------+------+-----+-------------------+
| film_id              | smallint(5)
unsigned                     | NO   | PRI |
NULL               |
| title                |
varchar(255)                          | NO   | MUL |
NULL               |
| description          |
text                                  | YES  |     |
NULL               |
| release_year         |
year(4)                               | YES  |     |
NULL               |
| language_id          | tinyint(3)
unsigned                      | NO   | MUL |
NULL               |
| original_language_id | tinyint(3)
unsigned                      | YES  | MUL |
NULL               |
| rental_duration      | tinyint(3)
unsigned                      | NO   |     |
3                  |
| rental_rate          |
decimal(4,2)                          | NO   |     |
4.99               |
| length               | smallint(5)
unsigned                     | YES  |     |
NULL               |
| replacement_cost     |
decimal(5,2)                          | NO   |     |
19.99              |
| rating               | enum('G','PG','PG-13','R','NC-
17')       | YES  |     | G                  |
| special_features     | set('Trailers',...,'Behind the
```

```
Scenes')  | YES  |     | NULL              |
| last_update         |
timestamp                                | NO   |     |
CURRENT_TIMESTAMP |
| prequel_film_id      | smallint(5)
unsigned                     | YES  | MUL |
NULL              |
+----------------------+------------------------------------
------+------+-----+-------------------+
```

Using a *self-join*, you can write a query that lists every film which has a prequel, along with the prequel's title:

```
mysql> SELECT f.title, f_prnt.title prequel
    -> FROM film f
    ->   INNER JOIN film f_prnt
    ->   ON f_prnt.film_id = f.prequel_film_id
    -> WHERE f.prequel_film_id IS NOT NULL;
+-----------------+--------------+
| title           | prequel      |
+-----------------+--------------+
| FIDDLER LOST II | FIDDLER LOST |
+-----------------+--------------+
1 row in set (0.00 sec)
```

This query joins the `film` table to itself using the `prequel_film_id` foreign key, and the table aliases `f` and `f_prnt` are assigned in order to make it clear which table is used for which purpose.

Test Your Knowledge

The following exercises are designed to test your understanding of inner joins. Please see Appendix C for the solutions to these exercises.

Exercise 5-1

Fill in the blanks (denoted by <#>) for the following query to obtain the results that follow:

```
mysql> SELECT c.first_name, c.last_name, a.address, ct.city
    -> FROM customer c
    ->   INNER JOIN address <1>
    ->   ON c.address_id = a.address_id
    ->   INNER JOIN city ct
    ->   ON a.city_id = <2>
    -> WHERE a.district = 'California';
+------------+-----------+------------------------+---------
-------+
| first_name | last_name | address                |
city           |
+------------+-----------+------------------------+---------
-------+
| PATRICIA   | JOHNSON   | 1121 Loja Avenue       | San
Bernardino |
| BETTY      | WHITE     | 770 Bydgoszcz Avenue   | Citrus
Heights |
| ALICE      | STEWART   | 1135 Izumisano Parkway |
Fontana        |
| ROSA       | REYNOLDS  | 793 Cam Ranh Avenue    |
Lancaster      |
| RENEE      | LANE      | 533 al-Ayn Boulevard   |
Compton        |
| KRISTIN    | JOHNSTON  | 226 Brest Manor        |
Sunnyvale      |
| CASSANDRA  | WALTERS   | 920 Kumbakonam Loop    |
Salinas        |
| JACOB      | LANCE     | 1866 al-Qatif Avenue   | El
Monte       |
| RENE       | MCALISTER | 1895 Zhezqazghan Drive | Garden
Grove   |
+------------+-----------+------------------------+---------
-------+
9 rows in set (0.00 sec)
```

Exercise 5-2

Write a query that returns the title of every film in which an actor with the first name JOHN appeared.

Exercise 5-3

Construct a query that finds returns all addresses which are in the same city. You will need to join the address table to itself, and each row should include 2 different addresses.

Chapter 6. Working with Sets

Although you can interact with the data in a database one row at a time, relational databases are really all about sets. This chapter explores how you can combine multiple result sets using various set operators. After a quick overview of set theory, I'll demonstrate how to use the set operators `union` , `intersect` , and `except` to blend multiple data sets together.

Set Theory Primer

In many parts of the world, basic set theory is included in elementary-level math curriculums. Perhaps you recall looking at something like what is shown in Figure 6-1.

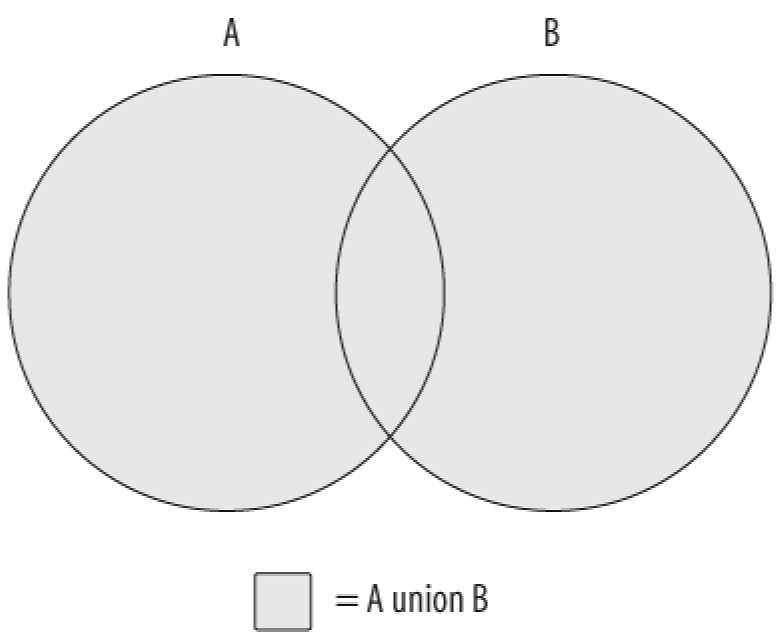

Figure 6-1. The union operation

The shaded area in Figure 6-1 represents the *union* of sets A and B, which is the combination of the two sets (with any overlapping regions included only once). Is this starting to look familiar? If so, then you'll finally get a chance to put that knowledge to use; if not, don't worry, because it's easy to visualize using a couple of diagrams.

Using circles to represent two data sets (A and B), imagine a subset of data that is common to both sets; this common data is represented by the overlapping area shown in Figure 6-1. Since set theory is rather uninteresting without an overlap between data sets, I use the same diagram to illustrate each set operation. There is another set operation that is concerned *only* with the overlap between two data sets; this operation is known as the *intersection* and is demonstrated in Figure 6-2.

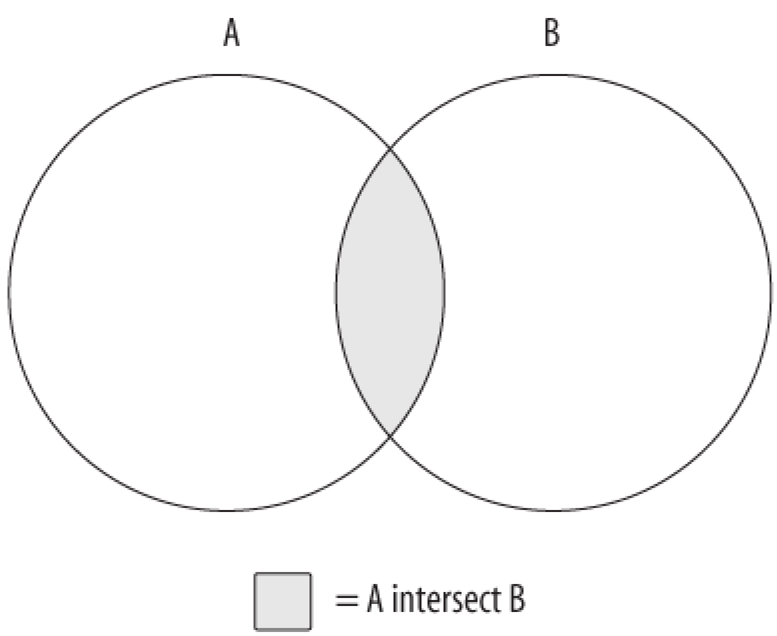

Figure 6-2. The intersection operation

The data set generated by the intersection of sets A and B is just the area of overlap between the two sets. If the two sets have no overlap, then the intersection operation yields the empty set.

The third and final set operation, which is demonstrated in Figure 6-3, is known as the *except* operation.

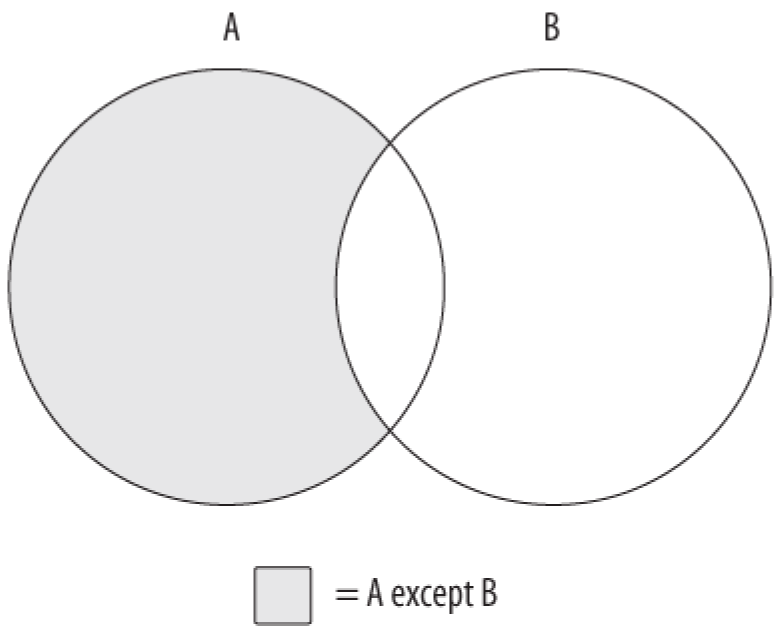

Figure 6-3. The except operation

Figure 6-3 shows the results of `A except B`, which is the whole of set A minus any overlap with set B. If the two sets have no overlap, then the operation `A except B` yields the whole of set A.

Using these three operations, or by combining different operations together, you can generate whatever results you need. For example, imagine that you want to build a set demonstrated by Figure 6-4.

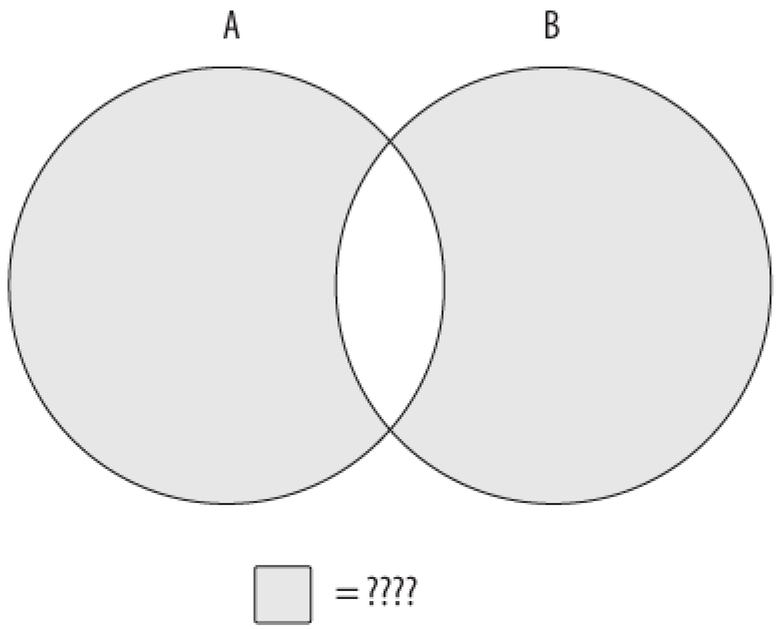

Figure 6-4. Mystery data set

The data set you are looking for includes all of sets A and B *without* the overlapping region. You can't achieve this outcome with just one of the three operations shown earlier; instead, you will need to first build a data set that encompasses all of sets A and B, and then utilize a second operation to remove the overlapping region. If the combined set is described as `A union B`, and the overlapping region is described as `A intersect B`, then the operation needed to generate the data set represented by Figure 6-4 would look as follows:

```
(A union B) except (A intersect B)
```

Of course, there are often multiple ways to achieve the same results; you could reach a similar outcome using the following operation:

```
(A except B) union (B except A)
```

While these concepts are fairly easy to understand using diagrams, the next sections show you how these concepts are applied to a relational database using the SQL set operators.

Set Theory in Practice

The circles used in the previous section's diagrams to represent data sets don't convey anything about what the data sets comprise. When dealing with actual data, however, there is a need to describe the composition of the data sets involved if they are to be combined. Imagine, for example, what would happen if you tried to generate the union of the `customer` table and the `city` table, whose definitions are as follows:

```
mysql> desc customer;
+-------------+----------------------+------+-----+---------
----------+
| Field       | Type                 | Null | Key |
Default           |
+-------------+----------------------+------+-----+---------
----------+
| customer_id | smallint(5) unsigned | NO   | PRI |
NULL              |
| store_id    | tinyint(3) unsigned  | NO   | MUL |
NULL              |
| first_name  | varchar(45)          | NO   |     |
NULL              |
| last_name   | varchar(45)          | NO   | MUL |
NULL              |
| email       | varchar(50)          | YES  |     |
NULL              |
| address_id  | smallint(5) unsigned | NO   | MUL |
NULL              |
| active      | tinyint(1)           | NO   |     |
1                 |
| create_date | datetime             | NO   |     |
NULL              |
| last_update | timestamp            | YES  |     |
```

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

mysql> desc city;
+-------------+----------------------+------+-----+---------
----------+
| Field       | Type                 | Null | Key |
Default           |
+-------------+----------------------+------+-----+---------
----------+
| city_id     | smallint(5) unsigned | NO   | PRI |
NULL              |
| city        | varchar(50)          | NO   |     |
NULL              |
| country_id  | smallint(5) unsigned | NO   | MUL |
NULL              |
| last_update | timestamp            | NO   |     |
CURRENT_TIMESTAMP |
+-------------+----------------------+------+-----+---------
----------+
```

When combined, the first column in the result set would include both the `customer.customer_id` and `city.city_id` columns, the second column would be the combination of the `customer.store_id` and `city.city` columns, and so forth. While some of the column pairs are easy to combine (e.g., two numeric columns), it is unclear how other column pairs should be combined, such as a numeric column with a string column or a string column with a date column. Additionally, the fifth through ninth columns of the combined tables would include data from only the `customer` table's fifth through ninth columns, since the `city` table has only four columns. Clearly, there needs to be some commonality between two data sets that you wish to combine.

Therefore, when performing set operations on two data sets, the following guidelines must apply:

- Both data sets must have the same number of columns.
- The data types of each column across the two data sets must be the same (or the server must be able to convert one to the other).

With these rules in place, it is easier to envision what “overlapping data” means in practice; each column pair from the two sets being combined must contain the same string, number, or date for rows in the two tables to be considered the same.

You perform a set operation by placing a *set operator* between two `select` statements, as demonstrated by the following:

```
mysql> SELECT 1 num, 'abc' str
    -> UNION
    -> SELECT 9 num, 'xyz' str;
+-----+-----+
| num | str |
+-----+-----+
|   1 | abc |
|   9 | xyz |
+-----+-----+
2 rows in set (0.02 sec)
```

Each of the individual queries yields a data set consisting of a single row having a numeric column and a string column. The set operator, which in this case is `union`, tells the database server to combine all rows from the two sets. Thus, the final set includes two rows of two columns. This query is known as a *compound query* because it comprises multiple, otherwise-independent queries. As you will see later, compound queries may include *more* than two queries if multiple set operations are needed to attain the final results.

Set Operators

The SQL language includes three set operators that allow you to perform each of the various set operations described earlier in the chapter. Additionally, each set operator has two flavors, one that includes duplicates and another that removes duplicates (but not necessarily *all* of the duplicates). The following subsections define each operator and demonstrate how they are used.

The union Operator

The `union` and `union all` operators allow you to combine multiple data sets. The difference between the two is that `union` sorts the combined set and removes duplicates, whereas `union all` does not. With `union all`, the number of rows in the final data set will always equal the sum of the number of rows in the sets being combined. This operation is the simplest set operation to perform (from the server's point of view), since there is no need for the server to check for overlapping data. The following example demonstrates how you can use the `union all` operator to generate a set of first and last names from multiple tables:

```
mysql> SELECT 'CUST' typ, c.first_name, c.last_name
    -> FROM customer c
    -> UNION ALL
    -> SELECT 'ACTR' typ, a.first_name, a.last_name
    -> FROM actor a;
+------+------------+-------------+
| typ  | first_name | last_name   |
+------+------------+-------------+
| CUST | MARY       | SMITH       |
| CUST | PATRICIA   | JOHNSON     |
| CUST | LINDA      | WILLIAMS    |
| CUST | BARBARA    | JONES       |
```

```
| CUST | ELIZABETH  | BROWN       |
| CUST | JENNIFER   | DAVIS       |
| CUST | MARIA      | MILLER      |
| CUST | SUSAN      | WILSON      |
| CUST | MARGARET   | MOORE       |
| CUST | DOROTHY    | TAYLOR      |
| CUST | LISA       | ANDERSON    |
| CUST | NANCY      | THOMAS      |
| CUST | KAREN      | JACKSON     |
...
| ACTR | BURT       | TEMPLE      |
| ACTR | MERYL      | ALLEN       |
| ACTR | JAYNE      | SILVERSTONE |
| ACTR | BELA       | WALKEN      |
| ACTR | REESE      | WEST        |
| ACTR | MARY       | KEITEL      |
| ACTR | JULIA      | FAWCETT     |
| ACTR | THORA      | TEMPLE      |
+------+------------+-------------+
799 rows in set (0.00 sec)
```

The query returns 799 names, with 599 rows coming from the `customer` table and the other 200 coming from the `actor` table. The first column, which has the alias `typ`, is not necessary, but was added to show the source of each name returned by the query.

Just to drive home the point that the `union all` operator doesn't remove duplicates, here's another version of the previous example, but with two identical queries against the `actor` table:

```
mysql> SELECT 'ACTR' typ, a.first_name, a.last_name
    -> FROM actor a
    -> UNION ALL
    -> SELECT 'ACTR' typ, a.first_name, a.last_name
    -> FROM actor a;
+------+-------------+--------------+
| typ  | first_name  | last_name    |
+------+-------------+--------------+
| ACTR | PENELOPE    | GUINESS      |
| ACTR | NICK        | WAHLBERG     |
| ACTR | ED          | CHASE        |
```

```
| ACTR | JENNIFER    | DAVIS        |
| ACTR | JOHNNY      | LOLLOBRIGIDA |
| ACTR | BETTE       | NICHOLSON    |
| ACTR | GRACE       | MOSTEL       |
...
| ACTR | BURT        | TEMPLE       |
| ACTR | MERYL       | ALLEN        |
| ACTR | JAYNE       | SILVERSTONE  |
| ACTR | BELA        | WALKEN       |
| ACTR | REESE       | WEST         |
| ACTR | MARY        | KEITEL       |
| ACTR | JULIA       | FAWCETT      |
| ACTR | THORA       | TEMPLE       |
+------+-------------+--------------+
400 rows in set (0.00 sec)
```

As you can see by the results, the 200 rows from the `actor` table are included twice, for a total of 400 rows.

While you are unlikely to repeat the same query twice in a compound query, here is another compound query that returns duplicate data:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE
'D%'
    -> UNION ALL
    -> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE
'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
| JENNIFER   | DAVIS     |
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
+------------+-----------+
5 rows in set (0.00 sec)
```

Both queries return the names of people having the initials “JD”. Of the five rows in the result set, one of them is a duplicate (Jennifer Davis). If you would like your combined table to *exclude* duplicate rows, you need to use the `union` operator instead of `union all`:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE
'D%'
    -> UNION
    -> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE
'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
+------------+-----------+
4 rows in set (0.00 sec)
```

For this version of the query, only the four distinct names are included in the result set, rather than the five rows returned when using `union all`.

The intersect Operator

The ANSI SQL specification includes the `intersect` operator for performing intersections. Unfortunately, version 8.0 of MySQL does not implement the `intersect` operator. If you are using Oracle or SQL Server 2008, you will be able to use `intersect`; since I am using MySQL for all examples in this book, however, the result sets for the example queries in this section are fabricated and cannot be executed with any versions up to and including version 8.0. I also refrain from showing the MySQL prompt (`mysql>`), since the statements are not being executed by the MySQL server.

If the two queries in a compound query return nonoverlapping data sets, then the intersection will be an empty set. Consider the following query:

```
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'D%' AND c.last_name LIKE 'T%'
INTERSECT
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'D%' AND a.last_name LIKE 'T%';
Empty set (0.04 sec)
```

While there are both actors and customers having the initials "DT", these sets are completely nonoverlapping, so the intersection of the two sets yields the empty set. If we switch back to the initials "JD", however, the intersection will yield a single row:

```
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
INTERSECT
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
+------------+-----------+
1 row in set (0.00 sec)
```

The intersection of these two queries yields Jennifer Davis, which is the only name found in both queries’ result sets.

Along with the `intersect` operator, which removes any duplicate rows found in the overlapping region, the ANSI SQL specification calls for an `intersect all` operator, which does not remove duplicates. The only database server that currently implements the `intersect all` operator is IBM’s DB2 Universal Server.

The except Operator

The ANSI SQL specification includes the `except` operator for performing the except operation. Once again, unfortunately, version 8.0 of MySQL does not implement the `except` operator, so the same rules apply for this section as for the previous section.

> **NOTE**
>
> If you are using Oracle Database, you will need to use the non-ANSI-compliant `minus` operator instead.

The `except` operator returns the first result set minus any overlap with the second result set. Here’s the example from the previous section, but using `except` instead of `intersect`, and with the order of the queries reversed:

```
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
EXCEPT
```

```
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
+------------+-----------+
3 rows in set (0.00 sec)
```

In this version of the query, the result set consists of the three rows from the first query minus Jennifer Davis, who is found in the result sets from both queries. There is also an `except all` operator specified in the ANSI SQL specification, but once again, only IBM's DB2 Universal Server has implemented the `except all` operator.

The `except all` operator is a bit tricky, so here's an example to demonstrate how duplicate data is handled. Let's say you have two data sets that look as follows:

- Set A

```
+----------+
| actor_id |
+----------+
|       10 |
|       11 |
|       12 |
|       10 |
|       10 |
+----------+
```

- Set B

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

```
|       10 |
|       10 |
+----------+
```

The operation `A except B` yields the following:

```
+----------+
| actor_id |
+----------+
|       11 |
|       12 |
+----------+
```

If you change the operation to `A except all B`, you will see the following:

```
+----------+
| actor_id |
+----------+
|       10 |
|       11 |
|       12 |
+----------+
```

Therefore, the difference between the two operations is that `except` removes all occurrences of duplicate data from set A, whereas `except all` only removes one occurrence of duplicate data from set A for every occurrence in set B.

Set Operation Rules

The following sections outline some rules that you must follow when working with compound queries.

Sorting Compound Query Results

If you want the results of your compound query to be sorted, you can add an `order by` clause after the last query. When specifying column names in the `order by` clause, you will need to choose from the column names in the first query of the compound query. Frequently, the column names are the same for both queries in a compound query, but this does not need to be the case, as demonstrated by the following:

```
mysql> SELECT a.first_name fname, a.last_name lname
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE
'D%'
    -> UNION ALL
    -> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE
'D%'
    -> ORDER BY lname, fname;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| JENNIFER | DAVIS     |
| JENNIFER | DAVIS     |
| JUDY     | DEAN      |
| JODIE    | DEGENERES |
| JULIANNE | DENCH     |
+----------+-----------+
5 rows in set (0.00 sec)
```

The column names specified in the two queries are different in this example. If you specify a column name from the second query in your `order by` clause, you will see the following error:

```
mysql> SELECT a.first_name fname, a.last_name lname
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE
'D%'
    -> UNION ALL
    -> SELECT c.first_name, c.last_name
```

```
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE
'D%'
    -> ORDER BY last_name, first_name;
ERROR 1054 (42S22): Unknown column 'last_name' in 'order
clause'
```

I recommend giving the columns in both queries identical column aliases in order to avoid this issue.

Set Operation Precedence

If your compound query contains more than two queries using different set operators, you need to think about the order in which to place the queries in your compound statement to achieve the desired results. Consider the following three-query compound statement:

```
mysql> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE
'D%'
    -> UNION ALL
    -> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'M%' AND a.last_name LIKE
'T%'
    -> UNION
    -> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE
'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
| MARY       | TANDY     |
| MENA       | TEMPLE    |
+------------+-----------+
6 rows in set (0.00 sec)
```

This compound query includes three queries that return sets of nonunique names; the first and second queries are separated with the `union all` operator, while the second and third queries are separated with the `union` operator. While it might not seem to make much difference where the `union` and `union all` operators are placed, it does, in fact, make a difference. Here's the same compound query with the set operators reversed:

```
mysql> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE
'D%'
    -> UNION
    -> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'M%' AND a.last_name LIKE
'T%'
    -> UNION ALL
    -> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE
'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
| MARY       | TANDY     |
| MENA       | TEMPLE    |
| JENNIFER   | DAVIS     |
+------------+-----------+
7 rows in set (0.00 sec)
```

Looking at the results, it's obvious that it *does* make a difference how the compound query is arranged when using different set operators. In general, compound queries containing three or more queries are evaluated in order from top to bottom, but with the following caveats:

- The ANSI SQL specification calls for the `intersect` operator to have precedence over the other set operators.
- You may dictate the order in which queries are combined by enclosing multiple queries in parentheses.

MySQL does not yet allow parentheses in compound queries, but if you are using a different database server, you can wrap adjoining queries in parentheses to override the default top-to-bottom processing of compound queries, as in:

```
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
UNION
(SELECT a.first_name, a.last_name
 FROM actor a
 WHERE a.first_name LIKE 'M%' AND a.last_name LIKE 'T%'
 UNION ALL
 SELECT c.first_name, c.last_name
 FROM customer c
 WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
)
```

For this compound query, the second and third queries would be combined using the `union all` operator, then the results would be combined with the first query using the `union` operator.

Test Your Knowledge

The following exercises are designed to test your understanding of set operations. See Appendix C for answers to these exercises.

Exercise 6-1

If set A = {L M N O P} and set B = {P Q R S T}, what sets are generated by the following operations?

- `A union B`
- `A union all B`
- `A intersect B`
- `A except B`

Exercise 6-2

Write a compound query that finds the first and last names of all Actors and Customers whose last name starts with L.

Exercise 6-3

Sort the results from Exercise 6-2 by the `last_name` column.

Chapter 7. Data Generation, Manipulation, and Conversion

As I mentioned in the Preface, this book strives to teach generic SQL techniques that can be applied across multiple database servers. This chapter, however, deals with the generation, conversion, and manipulation of string, numeric, and temporal data, and the SQL language does not include commands covering this functionality. Rather, built-in functions are used to facilitate data generation, conversion, and manipulation, and while the SQL standard does specify some functions, the database vendors often do not comply with the function specifications.

Therefore, my approach for this chapter is to show you some of the common ways in which data is generated and manipulated within SQL statements, and then demonstrate some of the built-in functions implemented by Microsoft SQL Server, Oracle Database, and MySQL. Along with reading this chapter, I strongly recommend you download a reference guide covering all the functions implemented by your server. If you work with more than one database server, there are several reference guides that cover multiple servers, such as Kevin Kline et al.'s *SQL in a Nutshell* and Jonathan Gennick's *SQL Pocket Guide*, both from O'Reilly.

Working with String Data

When working with string data, you will be using one of the following character data types:

`CHAR`

Holds fixed-length, blank-padded strings. MySQL allows `CHAR` values up to 255 characters in length, Oracle Database permits up to 2,000 characters, and SQL Server allows up to 8,000 characters.

`varchar`

Holds variable-length strings. MySQL permits up to 65,535 characters in a `varchar` column, Oracle Database (via the `varchar2` type) allows up to 4,000 characters, and SQL Server allows up to 8,000 characters.

`text` (MySQL and SQL Server) or `CLOB` (Character Large Object; Oracle Database)

Holds very large variable-length strings (generally referred to as documents in this context). MySQL has multiple text types (`tinytext`, `text`, `mediumtext`, and `longtext`) for documents up to 4 GB in size. SQL Server has a single `text` type for documents up to 2 GB in size, and Oracle Database includes the `CLOB` data type, which can hold documents up to a whopping 128 TB. SQL Server 2005 also includes the `varchar(max)` data type and recommends its use instead of the `text` type, which will be removed from the server in some future release.

To demonstrate how you can use these various types, I use the following table for some of the examples in this section:

```
CREATE TABLE string_tbl
 (char_fld CHAR(30),
  vchar_fld VARCHAR(30),
  text_fld TEXT
 );
```

The next two subsections show how you can generate and manipulate string data.

String Generation

The simplest way to populate a character column is to enclose a string in quotes, as in:

```
mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This is char data',
    ->   'This is varchar data',
    ->   'This is text data');
Query OK, 1 row affected (0.00 sec)
```

When inserting string data into a table, remember that if the length of the string exceeds the maximum size for the character column (either the designated maximum or the maximum allowed for the data type), the server will throw an exception. Although this is the default behavior for all three servers, you can configure MySQL and SQL Server to silently truncate the string instead of throwing an exception. To demonstrate how MySQL handles this situation, the following `update` statement attempts to modify the `vchar_fld` column, whose maximum length is defined as 30, with a string that is 46 characters in length:

```
mysql> UPDATE string_tbl
    -> SET vchar_fld = 'This is a piece of extremely long varchar data';
ERROR 1406 (22001): Data too long for column 'vchar_fld' at row 1
```

Since MySQL 6.0, the default behavior is now “strict” mode, which means that exceptions are thrown when problems arise, whereas in older versions of the server the string would have been truncated and a warning issued. If you would rather have the engine truncate the string and issue a warning instead of raising an exception, you can opt to be in ANSI mode. The following example shows how to check which mode you are in, and then how to change the mode using the `SET` command:

```
mysql> SELECT @@session.sql_mode;
+-----------------------------------------------------------------+
| @@session.sql_mode                                              |
+-----------------------------------------------------------------+
| STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |
+-----------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SET sql_mode='ansi';
Query OK, 0 rows affected (0.08 sec)

mysql> SELECT @@session.sql_mode;
+------------------------------------------------------------------------
-------+
| @@session.sql_mode
|
+------------------------------------------------------------------------
-------+
|
REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ONLY_FULL_GROUP_BY,ANSI
 |
+------------------------------------------------------------------------
-------+
1 row in set (0.00 sec)
```

If you rerun the previous `UPDATE` statement, you will find that the column has been modified, but the following warning is generated:

```
mysql> SHOW WARNINGS;
+---------+------+------------------------------------------------+
| Level   | Code | Message                                        |
+---------+------+------------------------------------------------+
| Warning | 1265 | Data truncated for column 'vchar_fld' at row 1 |
```

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

If you retrieve the `vchar_fld` column, you will see that the string has indeed been truncated:

```
mysql> SELECT vchar_fld
    -> FROM string_tbl;
+--------------------------------+
| vchar_fld                      |
+--------------------------------+
| This is a piece of extremely l |
+--------------------------------+
1 row in set (0.05 sec)
```

As you can see, only the first 30 characters of the 46-character string made it into the `vchar_fld` column. The best way to avoid string truncation (or exceptions, in the case of Oracle Database or MySQL in strict mode) when working with `varchar` columns is to set the upper limit of a column to a high enough value to handle the longest strings that might be stored in the column (keeping in mind that the server allocates only enough space to store the string, so it is not wasteful to set a high upper limit for `varchar` columns).

INCLUDING SINGLE QUOTES

Since strings are demarcated by single quotes, you will need to be alert for strings that include single quotes or apostrophes. For example, you won’t be able to insert the following string because the server will think that the apostrophe in the word *doesn’t* marks the end of the string:

```
UPDATE string_tbl
SET text_fld = 'This string doesn't work';
```

To make the server ignore the apostrophe in the word *doesn’t*, you will need to add an *escape* to the string so that the server treats the apostrophe like any other character in the string. All three servers allow you to escape a single quote by adding another single quote directly before, as in:

```
mysql> UPDATE string_tbl
    -> SET text_fld = 'This string didn''t work, but it does now';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

> **NOTE**
>
> Oracle Database and MySQL users may also choose to escape a single quote by adding a backslash character immediately before, as in:
>
> ```
> UPDATE string_tbl SET text_fld =
>   'This string didn\'t work, but it does now'
> ```

If you retrieve a string for use in a screen or report field, you don’t need to do anything special to handle embedded quotes:

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+-------------------------------------------+
| text_fld                                  |
+-------------------------------------------+
| This string didn't work, but it does now |
+-------------------------------------------+
1 row in set (0.00 sec)
```

However, if you are retrieving the string to add to a file that another program will read, you may want to include the escape as part of the retrieved string. If you are using MySQL, you can use the built-in function `quote()`, which places quotes around the entire string *and* adds escapes to any single quotes/apostrophes within the string. Here’s what our string looks like when retrieved via the `quote()` function:

```
mysql> SELECT quote(text_fld)
    -> FROM string_tbl;
+---------------------------------------------+
| QUOTE(text_fld)                             |
+---------------------------------------------+
| 'This string didn\'t work, but it does now' |
+---------------------------------------------+
1 row in set (0.04 sec)
```

When retrieving data for data export, you may want to use the `quote()` function for all non-system-generated character columns, such as a `customer_notes` column.

INCLUDING SPECIAL CHARACTERS

If your application is multinational in scope, you might find yourself working with strings that include characters that do not appear on your keyboard. When working with the French and German languages, for example, you might need to include accented characters such as é and ö. The SQL Server and MySQL servers include the built-in function `char()` so that you can build strings from any of the 255 characters in the ASCII character set (Oracle Database users can use the `chr()` function). To demonstrate, the next example retrieves a typed string and its equivalent built via individual characters:

```
mysql> SELECT 'abcdefg', CHAR(97,98,99,100,101,102,103);
+---------+--------------------------------+
| abcdefg | CHAR(97,98,99,100,101,102,103) |
+---------+--------------------------------+
| abcdefg | abcdefg                        |
+---------+--------------------------------+
1 row in set (0.01 sec)
```

Thus, the 97th character in the ASCII character set is the letter *a*. While the characters shown in the preceding example are not special, the following examples show the location of the accented characters along with other special characters, such as currency symbols:

```
mysql> SELECT CHAR(128,129,130,131,132,133,134,135,136,137);
+-----------------------------------------------+
| CHAR(128,129,130,131,132,133,134,135,136,137) |
+-----------------------------------------------+
| Çüéâäàåçêë                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(138,139,140,141,142,143,144,145,146,147);
+-----------------------------------------------+
| CHAR(138,139,140,141,142,143,144,145,146,147) |
+-----------------------------------------------+
| èïîìÄÅÉæÆô                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(148,149,150,151,152,153,154,155,156,157);
+-----------------------------------------------+
| CHAR(148,149,150,151,152,153,154,155,156,157) |
+-----------------------------------------------+
| öòûùÿÖÜø£Ø                                    |
+-----------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CHAR(158,159,160,161,162,163,164,165);
+-------------------------------------+
| CHAR(158,159,160,161,162,163,164,165) |
+-------------------------------------+
| ×ƒáíóúñÑ                            |
+-------------------------------------+
1 row in set (0.01 sec)
```

NOTE

I am using the utf8mb4 character set for the examples in this section. If your session is configured for a different character set, you will see a different set of characters than what is shown here. The same concepts apply, but you will need to familiarize yourself with the layout of your character set to locate specific characters.

Building strings character by character can be quite tedious, especially if only a few of the characters in the string are accented. Fortunately, you can use the `concat()` function to concatenate individual strings, some of which you can type while others you can generate via the `char()` function. For example, the following shows how to build the phrase *danke schön* using the `concat()` and `char()` functions:

```
mysql> SELECT CONCAT('danke sch', CHAR(148), 'n');
+-------------------------------------+
| CONCAT('danke sch', CHAR(148), 'n') |
+-------------------------------------+
| danke schön                         |
+-------------------------------------+
1 row in set (0.00 sec)
```

NOTE

Oracle Database users can use the concatenation operator (`||`) instead of the `concat()` function, as in:

```
SELECT 'danke sch' || CHR(148) || 'n'
FROM dual;
```

SQL Server does not include a concat() function, so you will need to use the concatenation operator (+), as in:

```
SELECT 'danke sch' + CHAR(148) + 'n'
```

If you have a character and need to find its ASCII equivalent, you can use the `ascii()` function, which takes the leftmost character in the string and returns a number:

```
mysql> SELECT ASCII('ö');
+------------+
| ASCII('ö') |
+------------+
|        148 |
```

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

Using the `char()`, `ascii()`, and `concat()` functions (or concatenation operators), you should be able to work with any Roman language even if you are using a keyboard that does not include accented or special characters.

String Manipulation

Each database server includes many built-in functions for manipulating strings. This section explores two types of string functions: those that return numbers and those that return strings. Before I begin, however, I reset the data in the `string_tbl` table to the following:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This string is 28 characters',
    ->   'This string is 28 characters',
    ->   'This string is 28 characters');
Query OK, 1 row affected (0.00 sec)
```

STRING FUNCTIONS THAT RETURN NUMBERS

Of the string functions that return numbers, one of the most commonly used is the `length()` function, which returns the number of characters in the string (SQL Server users will need to use the `len()` function). The following query applies the `length()` function to each column in the `string_tbl` table:

```
mysql> SELECT LENGTH(char_fld) char_length,
    ->   LENGTH(vchar_fld) varchar_length,
    ->   LENGTH(text_fld) text_length
    -> FROM string_tbl;
+-------------+----------------+-------------+
| char_length | varchar_length | text_length |
+-------------+----------------+-------------+
|          28 |             28 |          28 |
+-------------+----------------+-------------+
1 row in set (0.00 sec)
```

While the lengths of the `varchar` and `text` columns are as expected, you might have expected the length of the `char` column to be 30, since I told you that strings stored in `char` columns are right-padded with spaces. The MySQL server removes trailing spaces from `char` data when it is retrieved, however, so you will see the same results from all string functions regardless of the type of column in which the strings are stored.

Along with finding the length of a string, you might want to find the location of a substring within a string. For example, if you want to find the position at which the string `'characters'` appears in the `vchar_fld` column, you could use the `position()` function, as demonstrated by the following:

```
mysql> SELECT POSITION('characters' IN vchar_fld)
    -> FROM string_tbl;
+-------------------------------------+
| POSITION('characters' IN vchar_fld) |
+-------------------------------------+
|                                  19 |
+-------------------------------------+
1 row in set (0.12 sec)
```

If the substring cannot be found, the `position()` function returns `0`.

> **WARNING**
>
> For those of you who program in a language such as C or C++, `where the first element of an array is at position 0, remember when working with databases that the first character in a string is at position 1. A return value of 0 from` instr() indicates that the substring could not be found, not that the substring was found at the first position in the string.

If you want to start your search at something other than the first character of your target string, you will need to use the `locate()` function, which is similar to the `position()` function except that it allows an optional third parameter, which is used to define the search's start position. The `locate()` function is also proprietary, whereas the `position()` function is part of the SQL:2003 standard. Here's an example asking for the position of the string `'is'` starting at the fifth character in the `vchar_fld` column:

```
mysql> SELECT LOCATE('is', vchar_fld, 5)
    -> FROM string_tbl;
+----------------------------+
```

```
| LOCATE('is', vchar_fld, 5) |
+----------------------------+
|                         13 |
+----------------------------+
1 row in set (0.02 sec)
```

NOTE

Oracle Database does not include the `position()` or `locate()` function, but it does include the `instr()` function, which mimics the `position()` function when provided with two arguments and mimics the `locate()` function when provided with three arguments. SQL Server also doesn't include a `position()` or `locate()` function, but it does include the `charindx()` function, which also accepts either two or three arguments similar to Oracle's `instr()` function.

Another function that takes strings as arguments and returns numbers is the string comparison function `strcmp()`. `Strcmp()`, which is implemented only by MySQL and has no analog in Oracle Database or SQL Server, takes two strings as arguments, and returns one of the following:

- `−1` if the first string comes before the second string in sort order
- `0` if the strings are identical
- `1` if the first string comes after the second string in sort order

To illustrate how the function works, I first show the sort order of five strings using a query, and then show how the strings compare to one another using `strcmp()`. Here are the five strings that I insert into the `string_tbl` table:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld)
    -> VALUES ('abcd'),
    ->        ('xyz'),
    ->        ('QRSTUV'),
    ->        ('qrstuv'),
    ->        ('12345');
Query OK, 5 rows affected (0.05 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

Here are the five strings in their sort order:

```
mysql> SELECT vchar_fld
    -> FROM string_tbl
    -> ORDER BY vchar_fld;
+-----------+
| vchar_fld |
+-----------+
| 12345     |
| abcd      |
| QRSTUV    |
| qrstuv    |
| xyz       |
+-----------+
5 rows in set (0.00 sec)
```

The next query makes six comparisons among the five different strings:

```
mysql> SELECT STRCMP('12345','12345') 12345_12345,
    ->   STRCMP('abcd','xyz') abcd_xyz,
    ->   STRCMP('abcd','QRSTUV') abcd_QRSTUV,
    ->   STRCMP('qrstuv','QRSTUV') qrstuv_QRSTUV,
    ->   STRCMP('12345','xyz') 12345_xyz,
    ->   STRCMP('xyz','qrstuv') xyz_qrstuv;
+-------------+----------+-------------+---------------+-----------+------
------+
| 12345_12345 | abcd_xyz | abcd_QRSTUV | qrstuv_QRSTUV | 12345_xyz |
xyz_qrstuv |
+-------------+----------+-------------+---------------+-----------+------
------+
|           0 |       −1 |          −1 |             0 |        −1 |
1 |
+-------------+----------+-------------+---------------+-----------+------
------+
1 row in set (0.00 sec)
```

The first comparison yields `0`, which is to be expected since I compared a string to itself. The fourth comparison also yields `0`, which is a bit surprising, since the strings are composed of the same letters, with one string all uppercase and the other all lowercase. The reason for this result is that MySQL's `strcmp()` function is case-insensitive, which is something to remember when using the function. The other four comparisons yield either `−1` or `1` depending on whether the first string comes before or after the second string in sort order. For example, `strcmp('abcd','xyz')` yields `−1`, since the string `'abcd'` comes before the string `'xyz'`.

Along with the `strcmp()` function, MySQL also allows you to use the `like` and `regexp` operators to compare strings in the `select` clause. Such comparisons will yield `1` (for `true`) or `0` (for `false`). Therefore, these operators allow you to build expressions that return a number, much like the functions described in this section. Here's an example using `like`:

```
mysql> SELECT name, name LIKE '%y' ends_in_y
    -> FROM category;
+-------------+-----------+
| name        | ends_in_y |
+-------------+-----------+
| Action      |         0 |
| Animation   |         0 |
| Children    |         0 |
| Classics    |         0 |
| Comedy      |         1 |
| Documentary |         1 |
| Drama       |         0 |
| Family      |         1 |
| Foreign     |         0 |
| Games       |         0 |
| Horror      |         0 |
| Music       |         0 |
| New         |         0 |
| Sci-Fi      |         0 |
| Sports      |         0 |
| Travel      |         0 |
+-------------+-----------+
16 rows in set (0.00 sec)
```

This example retrieves all the category names, along with an expression that returns `1` if the name ends in "y" or `0` otherwise. If you want to perform more complex pattern matches, you can use the `regexp` operator, as demonstrated by the following:

```
mysql> SELECT name, name REGEXP 'y$' ends_in_y
    -> FROM category;
+-------------+-----------+
| name        | ends_in_y |
+-------------+-----------+
| Action      |         0 |
| Animation   |         0 |
| Children    |         0 |
| Classics    |         0 |
| Comedy      |         1 |
| Documentary |         1 |
| Drama       |         0 |
| Family      |         1 |
| Foreign     |         0 |
| Games       |         0 |
| Horror      |         0 |
| Music       |         0 |
| New         |         0 |
| Sci-Fi      |         0 |
| Sports      |         0 |
| Travel      |         0 |
```

```
+-------------+----------+
16 rows in set (0.00 sec)
```

The second column of this query returns `1` if the value stored in the `name` column matches the given regular expression.

> **NOTE**
>
> SQL Server and Oracle Database users can achieve similar results by building `case` expressions, which I describe in detail in Chapter 11.

STRING FUNCTIONS THAT RETURN STRINGS

In some cases, you will need to modify existing strings, either by extracting part of the string or by adding additional text to the string. Every database server includes multiple functions to help with these tasks. Before I begin, I once again reset the data in the `string_tbl` table:

```
mysql> DELETE FROM string_tbl;
Query OK, 5 rows affected (0.00 sec)

mysql> INSERT INTO string_tbl (text_fld)
    -> VALUES ('This string was 29 characters');
Query OK, 1 row affected (0.01 sec)
```

Earlier in the chapter, I demonstrated the use of the `concat()` function to help build words that include accented characters. The `concat()` function is useful in many other situations, including when you need to append additional characters to a stored string. For instance, the following example modifies the string stored in the `text_fld` column by tacking an additional phrase on the end:

```
mysql> UPDATE string_tbl
    -> SET text_fld = CONCAT(text_fld, ', but now it is longer');
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The contents of the `text_fld` column are now as follows:

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+-------------------------------------------------------+
| text_fld                                              |
```

```
+-------------------------------------------------+
| This string was 29 characters, but now it is longer |
+-------------------------------------------------+
1 row in set (0.00 sec)
```

Thus, like all functions that return a string, you can use `concat()` to replace the data stored in a character column.

Another common use for the `concat()` function is to build a string from individual pieces of data. For example, the following query generates a narrative string for each customer:

```
mysql> SELECT concat(first_name, ' ', last_name,
    ->   ' has been a customer since ', date(create_date)) cust_narrative
    -> FROM customer;
+-----------------------------------------------------------+
| cust_narrative                                            |
+-----------------------------------------------------------+
| MARY SMITH has been a customer since 2006-02-14           |
| PATRICIA JOHNSON has been a customer since 2006-02-14     |
| LINDA WILLIAMS has been a customer since 2006-02-14       |
| BARBARA JONES has been a customer since 2006-02-14        |
| ELIZABETH BROWN has been a customer since 2006-02-14      |
| JENNIFER DAVIS has been a customer since 2006-02-14       |
| MARIA MILLER has been a customer since 2006-02-14         |
| SUSAN WILSON has been a customer since 2006-02-14         |
| MARGARET MOORE has been a customer since 2006-02-14       |
| DOROTHY TAYLOR has been a customer since 2006-02-14       |
...
| RENE MCALISTER has been a customer since 2006-02-14       |
| EDUARDO HIATT has been a customer since 2006-02-14        |
| TERRENCE GUNDERSON has been a customer since 2006-02-14   |
| ENRIQUE FORSYTHE has been a customer since 2006-02-14     |
| FREDDIE DUGGAN has been a customer since 2006-02-14       |
| WADE DELVALLE has been a customer since 2006-02-14        |
| AUSTIN CINTRON has been a customer since 2006-02-14       |
+-----------------------------------------------------------+
599 rows in set (0.00 sec)
```

The `concat()` function can handle any expression that returns a string, and will even convert numbers and dates to string format, as evidenced by the date column (`create_date`) used as an argument. Although Oracle Database includes the `concat()` function, it will accept only two string arguments, so the previous query will not work on Oracle. Instead, you would need to use the concatenation operator (`||`) rather than a function call, as in:

```
SELECT first_name || ' ' || last_name ||
  ' has been a customer since ' || date(create_date)) cust_narrative
```

```
FROM customer;
```

SQL Server does not include a `concat()` function, so you would need to use the same approach as the previous query, except that you would use SQL Server's concatenation operator (+) instead of `||`.

While `concat()` is useful for adding characters to the beginning or end of a string, you may also have a need to add or replace characters in the *middle* of a string. All three database servers provide functions for this purpose, but all of them are different, so I demonstrate the MySQL function and then show the functions from the other two servers.

MySQL includes the `insert()` function, which takes four arguments: the original string, the position at which to start, the number of characters to replace, and the replacement string. Depending on the value of the third argument, the function may be used to either insert or replace characters in a string. With a value of `0` for the third argument, the replacement string is inserted and any trailing characters are pushed to the right, as in:

```
mysql> SELECT INSERT('goodbye world', 9, 0, 'cruel ') string;
+---------------------+
| string              |
+---------------------+
| goodbye cruel world |
+---------------------+
1 row in set (0.00 sec)
```

In this example, all characters starting from position 9 are pushed to the right and the string `'cruel'` is inserted. If the third argument is greater than zero, then that number of characters is replaced with the replacement string, as in:

```
mysql> SELECT INSERT('goodbye world', 1, 7, 'hello') string;
+-------------+
| string      |
+-------------+
| hello world |
+-------------+
1 row in set (0.00 sec)
```

For this example, the first seven characters are replaced with the string `'hello'`. Oracle Database does not provide a single function with the flexibility of MySQL's `insert()` function, but Oracle does provide the `replace()` function, which is useful for replacing one substring with another. Here's the previous example reworked to use `replace()`:

```
SELECT REPLACE('goodbye world', 'goodbye', 'hello')
FROM dual;
```

All instances of the string `'goodbye'` will be replaced with the string `'hello'`, resulting in the string `'hello world'`. The `replace()` function will replace *every* instance of the search string with the replacement string, so you need to be careful that you don't end up with more replacements than you anticipated.

SQL Server also includes a `replace()` function with the same functionality as Oracle's, but SQL Server also includes a function called `stuff()` with similar functionality to MySQL's `insert()` function. Here's an example:

```
SELECT STUFF('hello world', 1, 5, 'goodbye cruel')
```

When executed, five characters are removed starting at position 1, and then the string `'goodbye cruel'` is inserted at the starting position, resulting in the string `'goodbye cruel world'`.

Along with inserting characters into a string, you may have a need to *extract* a substring from a string. For this purpose, all three servers include the `substring()` function (although Oracle Database's version is called `substr()`), which extracts a specified number of characters starting at a specified position. The following example extracts five characters from a string starting at the ninth position:

```
mysql> SELECT SUBSTRING('goodbye cruel world', 9, 5);
+----------------------------------------+
| SUBSTRING('goodbye cruel world', 9, 5) |
+----------------------------------------+
| cruel                                  |
+----------------------------------------+
1 row in set (0.00 sec)
```

Along with the functions demonstrated here, all three servers include many more built-in functions for manipulating string data. While many of them are designed for very specific purposes, such as generating the string equivalent of octal or hexadecimal numbers, there are many other general-purpose functions as well, such as functions that remove or add trailing spaces. For more information, consult your server's SQL reference guide, or a general-purpose SQL reference guide such as *SQL in a Nutshell* (O'Reilly).

Working with Numeric Data

Unlike string data (and temporal data, as you will see shortly), numeric data generation is quite straightforward. You can type a number, retrieve it from another column, or generate it via a calculation. All the usual arithmetic operators (`+`, `-`, `*`, `/`) are available for performing calculations, and parentheses may be used to dictate precedence, as in:

```
mysql> SELECT (37 * 59) / (78 - (8 * 6));
+---------------------------+
| (37 * 59) / (78 - (8 * 6)) |
+---------------------------+
|                     72.77 |
+---------------------------+
1 row in set (0.00 sec)
```

As I mentioned in Chapter 2, the main concern when storing numeric data is that numbers might be rounded if they are larger than the specified size for a numeric column. For example, the number 9.96 will be rounded to 10.0 if stored in a column defined as `float(3,1)`.

Performing Arithmetic Functions

Most of the built-in numeric functions are used for specific arithmetic purposes, such as determining the square root of a number. Table 7-1 lists some of the common numeric functions that take a single numeric argument and return a number.

Table 7-1. Single-argument numeric functions

Function name	Description
`Acos( x )`	Calculates the arc cosine of *x*
`Asin( x )`	Calculates the arc sine of *x*
`Atan( x )`	Calculates the arc tangent of *x*
`Cos( x )`	Calculates the cosine of *x*
`Cot( x )`	Calculates the cotangent of *x*
`Exp( x )`	Calculates e^x
`Ln( x )`	Calculates the natural log of *x*
`Sin( x )`	Calculates the sine of *x*

Function name	Description
`Sqrt( x )`	Calculates the square root of *x*
`Tan( x )`	Calculates the tangent of *x*

These functions perform very specific tasks, and I refrain from showing examples for these functions (if you don't recognize a function by name or description, then you probably don't need it). Other numeric functions used for calculations, however, are a bit more flexible and deserve some explanation.

For example, the `modulo` operator, which calculates the remainder when one number is divided into another number, is implemented in MySQL and Oracle Database via the `mod()` function. The following example calculates the remainder when `4` is divided into `10`:

```
mysql> SELECT MOD(10,4);
+-----------+
| MOD(10,4) |
+-----------+
|         2 |
+-----------+
1 row in set (0.02 sec)
```

While the `mod()` function is typically used with integer arguments, with MySQL you can also use real numbers, as in:

```
mysql> SELECT MOD(22.75, 5);
+---------------+
| MOD(22.75, 5) |
+---------------+
|          2.75 |
+---------------+
1 row in set (0.02 sec)
```

NOTE

SQL Server does not have a `mod()` function. Instead, the operator `%` is used for finding remainders. The expression `10 % 4` will therefore yield the value `2`.

Another numeric function that takes two numeric arguments is the `pow()` function (or `power()` if you are using Oracle Database or SQL Server), which returns one number raised to the power of a second number, as in:

```
mysql> SELECT POW(2,8);
+----------+
| POW(2,8) |
+----------+
|      256 |
+----------+
1 row in set (0.03 sec)
```

Thus, `pow(2,8)` is the MySQL equivalent of specifying 2^8. Since computer memory is allocated in chunks of 2^x bytes, the `pow()` function can be a handy way to determine the exact number of bytes in a certain amount of memory:

```
mysql> SELECT POW(2,10) kilobyte, POW(2,20) megabyte,
    ->   POW(2,30) gigabyte, POW(2,40) terabyte;
+----------+----------+------------+---------------+
| kilobyte | megabyte | gigabyte   | terabyte      |
+----------+----------+------------+---------------+
|     1024 |  1048576 | 1073741824 | 1099511627776 |
+----------+----------+------------+---------------+
1 row in set (0.00 sec)
```

I don’t know about you, but I find it easier to remember that a gigabyte is 2^{30} bytes than to remember the number 1,073,741,824.

Controlling Number Precision

When working with floating-point numbers, you may not always want to interact with or display a number with its full precision. For example, you may store monetary transaction data with a precision to six decimal places, but you might want to round to the nearest hundredth for display purposes. Four functions are useful when limiting the precision of floating-point numbers: `ceil()`, `floor()`, `round()`, and `truncate()`. All three servers include these functions, although Oracle Database includes `trunc()` instead of `truncate()`, and SQL Server includes `ceiling()` instead of `ceil()`.

The `ceil()` and `floor()` functions are used to round either up or down to the closest integer, as demonstrated by the following:

```
mysql> SELECT CEIL(72.445), FLOOR(72.445);
+--------------+---------------+
| CEIL(72.445) | FLOOR(72.445) |
+--------------+---------------+
|           73 |            72 |
+--------------+---------------+
1 row in set (0.06 sec)
```

Thus, any number between 72 and 73 will be evaluated as 73 by the `ceil()` function and 72 by the `floor()` function. Remember that `ceil()` will round up even if the decimal portion of a number is very small, and `floor()` will round down even if the decimal portion is quite significant, as in:

```
mysql> SELECT CEIL(72.000000001), FLOOR(72.999999999);
+--------------------+---------------------+
| CEIL(72.000000001) | FLOOR(72.999999999) |
+--------------------+---------------------+
|                 73 |                  72 |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

If this is a bit too severe for your application, you can use the `round()` function to round up or down from the *midpoint* between two integers, as in:

```
mysql> SELECT ROUND(72.49999), ROUND(72.5), ROUND(72.50001);
+-----------------+-------------+-----------------+
| ROUND(72.49999) | ROUND(72.5) | ROUND(72.50001) |
+-----------------+-------------+-----------------+
|              72 |          73 |              73 |
+-----------------+-------------+-----------------+
1 row in set (0.00 sec)
```

Using `round()`, any number whose decimal portion is halfway or more between two integers will be rounded up, whereas the number will be rounded down if the decimal portion is anything less than halfway between the two integers.

Most of the time, you will want to keep at least some part of the decimal portion of a number rather than rounding to the nearest integer; the `round()` function allows an optional second argument to specify how many digits to the right of the decimal place to round to. The next example shows how you can use the second argument to round the number 72.0909 to one, two, and three decimal places:

```
mysql> SELECT ROUND(72.0909, 1), ROUND(72.0909, 2), ROUND(72.0909, 3);
+-------------------+-------------------+-------------------+
| ROUND(72.0909, 1) | ROUND(72.0909, 2) | ROUND(72.0909, 3) |
+-------------------+-------------------+-------------------+
|              72.1 |             72.09 |            72.091 |
+-------------------+-------------------+-------------------+
1 row in set (0.00 sec)
```

Like the `round()` function, the `truncate()` function allows an optional second argument to specify the number of digits to the right of the decimal, but `truncate()` simply discards the unwanted digits without rounding. The next example shows how the number 72.0909 would be truncated to one, two, and three decimal places:

```
mysql> SELECT TRUNCATE(72.0909, 1), TRUNCATE(72.0909, 2),
    ->   TRUNCATE(72.0909, 3);
+----------------------+----------------------+----------------------+
| TRUNCATE(72.0909, 1) | TRUNCATE(72.0909, 2) | TRUNCATE(72.0909, 3) |
+----------------------+----------------------+----------------------+
|                 72.0 |                72.09 |               72.090 |
+----------------------+----------------------+----------------------+
1 row in set (0.00 sec)
```

NOTE

SQL Server does not include a `truncate()` function. Instead, the `round()` function allows for an optional third argument which, if present and nonzero, calls for the number to be truncated rather than rounded.

Both `truncate()` and `round()` also allow a *negative* value for the second argument, meaning that numbers to the *left* of the decimal place are truncated or rounded. This might seem like a strange thing to do at first, but there are valid applications. For example, you might sell a product that can be purchased only in units of 10. If a customer were to order 17 units, you could choose from one of the following methods to modify the customer’s order quantity:

```
mysql> SELECT ROUND(17, −1), TRUNCATE(17, −1);
+---------------+------------------+
| ROUND(17, −1) | TRUNCATE(17, −1) |
+---------------+------------------+
|            20 |               10 |
+---------------+------------------+
1 row in set (0.00 sec)
```

If the product in question is thumbtacks, then it might not make much difference to your bottom line whether you sold the customer 10 or 20 thumbtacks when only 17 were requested; if you are selling Rolex watches, however, your business may fare better by rounding.

Handling Signed Data

If you are working with numeric columns that allow negative values (in Chapter 2, I showed how a numeric column may be labeled *unsigned*, meaning that only positive numbers are allowed), several numeric functions might be of use. Let's say, for example, that you are asked to generate a report showing the current status of a set of bank accounts using the following data from the `account` table:

```
+------------+--------------+---------+
| account_id | acct_type    | balance |
+------------+--------------+---------+
|        123 | MONEY MARKET |  785.22 |
|        456 | SAVINGS      |    0.00 |
|        789 | CHECKING     | -324.22 |
+------------+--------------+---------+
```

The following query returns three columns useful for generating the report:

```
mysql> SELECT account_id, SIGN(balance), ABS(balance)
    -> FROM account;
+------------+---------------+--------------+
| account_id | SIGN(balance) | ABS(balance) |
+------------+---------------+--------------+
|        123 |             1 |       785.22 |
|        456 |             0 |         0.00 |
|        789 |            -1 |       324.22 |
+------------+---------------+--------------+
3 rows in set (0.00 sec)
```

The second column uses the `sign()` function to return `−1` if the account balance is negative, `0` if the account balance is zero, and `1` if the account balance is positive.

The third column returns the absolute value of the account balance via the `abs()` function.

Working with Temporal Data

Of the three types of data discussed in this chapter (character, numeric, and temporal), temporal data is the most involved when it comes to data generation and manipulation. Some of the complexity of temporal data is caused by the myriad ways in which a single date and time can be described. For example, the date on which I wrote this paragraph can be described in all the following ways:

- Wednesday, June 5, 2019
- 6/05/2019 2:14:56 P.M. EST
- 6/05/2019 19:14:56 GMT
- 1562019 (Julian format)
- Star date [−4] 97026.79 14:14:56 (*Star Trek* format)

While some of these differences are purely a matter of formatting, most of the complexity has to do with your frame of reference, which we explore in the next section.

Dealing with Time Zones

Because people around the world prefer that noon coincides roughly with the sun's peak at their location, there has never been a serious attempt to coerce everyone to use a universal clock. Instead, the world has been sliced into 24 imaginary sections, called *time zones*; within a particular time zone, everyone agrees on the current time, whereas people in different time zones do not. While this seems simple enough, some geographic regions shift their time by one hour twice a year (implementing what is known as *daylight saving time*) and some do not, so the time difference between two points on Earth might be four hours for one half of the year and five hours for the other half of the year. Even within a single time zone, different regions may or may not adhere to daylight saving time, causing different clocks in the same time zone to agree for one half of the year but be one hour different for the rest of the year.

While the computer age has exacerbated the issue, people have been dealing with time zone differences since the early days of naval exploration. To ensure a common point of reference for timekeeping, fifteenth-century navigators set their clocks to the time of day in Greenwich, England. This became known as *Greenwich Mean Time*, or GMT. All other time zones can be described by the number of hours' difference from GMT; for example, the time zone for the Eastern United States, known as *Eastern Standard Time*, can be described as GMT −5:00, or five hours earlier than GMT.

Today, we use a variation of GMT called *Coordinated Universal Time*, or UTC, which is based on an atomic clock (or, to be more precise, the average time of 200 atomic clocks in 50 locations worldwide, which is referred to as *Universal Time*). Both SQL Server and MySQL provide functions that will return the current UTC timestamp (`getutcdate()` for SQL Server and `utc_timestamp()` for MySQL).

Most database servers default to the time zone setting of the server on which it resides and provide tools for modifying the time zone if needed. For example, a database used to store stock exchange transactions from around the world would generally be configured to use UTC time, whereas a database used to store transactions at a particular retail establishment might use the server's time zone.

MySQL keeps two different time zone settings: a global time zone, and a session time zone, which may be different for each user logged in to a database. You can see both settings via the following query:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | SYSTEM              |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

A value of `system` tells you that the server is using the time zone setting from the server on which the database resides.

If you are sitting at a computer in Zurich, Switzerland, and you open a session across the network to a MySQL server situated in New York, you may want to change the time zone setting for your session, which you can do via the following command:

```
mysql> SET time_zone = 'Europe/Zurich';
Query OK, 0 rows affected (0.18 sec)
```

If you check the time zone settings again, you will see the following:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | Europe/Zurich       |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

All dates displayed in your session will now conform to Zurich time.

> **NOTE**
>
> Oracle Database users can change the time zone setting for a session via the following command:
>
> ```
> ALTER SESSION TIMEZONE = 'Europe/Zurich'
> ```

Generating Temporal Data

You can generate temporal data via any of the following means:

- Copying data from an existing `date`, `datetime`, or `time` column
- Executing a built-in function that returns a `date`, `datetime`, or `time`
- Building a string representation of the temporal data to be evaluated by the server

To use the last method, you will need to understand the various components used in formatting dates.

STRING REPRESENTATIONS OF TEMPORAL DATA

Table 2-4 in Chapter 2 presented the more popular date components; to refresh your memory, Table 7-2 shows these same components.

Table 7-2. Date format components

Component	Definition	Range
YYYY	Year, including century	`1000` to `9999`
MM	Month	`01` (January) to `12` (December)
DD	Day	`01` to `31`
HH	Hour	`00` to `23`
HHH	Hours (elapsed)	`-838` to `838`
MI	Minute	`00` to `59`
SS	Second	`00` to `59`

To build a string that the server can interpret as a `date`, `datetime`, or `time`, you need to put the various components together in the order shown in Table 7-3.

Table 7-3. Required date components

Type	Default format
`Date`	YYYY-MM-DD
`Datetime`	YYYY-MM-DD HH:MI:SS
`Timestamp`	YYYY-MM-DD HH:MI:SS
`Time`	HHH:MI:SS

Thus, to populate a `datetime` column with 3:30 P.M. on September 17, 2019, you will need to build the following string:

```
'2019-09-17 15:30:00'
```

If the server is expecting a `datetime` value, such as when updating a `datetime` column or when calling a built-in function that takes a `datetime` argument, you can provide a properly formatted string with the required date components, and the server will do the conversion for you. For example, here's a statement used to modify the return date of a film rental:

```
UPDATE rental
SET return_date = '2019-09-17 15:30:00'
WHERE rental_id = 99999;
```

The server determines that the string provided in the `set` clause must be a `datetime` value, since the string is being used to populate a `datetime` column.

Therefore, the server will attempt to convert the string for you by parsing the string into the six components (year, month, day, hour, minute, second) included in the default `datetime` format.

STRING-TO-DATE CONVERSIONS

If the server is *not* expecting a `datetime` value, or if you would like to represent the `datetime` using a nondefault format, you will need to tell the server to convert the string to a `datetime`. For example, here is a simple query that returns a `datetime` value using the `cast()` function:

```
mysql> SELECT CAST('2019-09-17 15:30:00' AS DATETIME);
+-----------------------------------------+
| CAST('2019-09-17 15:30:00' AS DATETIME) |
+-----------------------------------------+
| 2019-09-17 15:30:00                     |
+-----------------------------------------+
1 row in set (0.00 sec)
```

We cover the `cast()` function at the end of this chapter. While this example demonstrates how to build `datetime` values, the same logic applies to the `date` and `time` types as well. The following query uses the `cast()` function to generate a `date` value and a `time` value:

```
mysql> SELECT CAST('2019-09-17' AS DATE) date_field,
    ->   CAST('108:17:57' AS TIME) time_field;
+------------+-----------+
| date_field | time_field |
+------------+-----------+
| 2019-09-17 | 108:17:57  |
+------------+-----------+
1 row in set (0.00 sec)
```

You may, of course, explicitly convert your strings even when the server is expecting a `date`, `datetime`, or `time` value, rather than letting the server do an implicit conversion.

When strings are converted to temporal values—whether explicitly or implicitly—you must provide all the date components in the required order. While some servers are quite strict regarding the date format, the MySQL server is quite lenient about the separators used between the components. For example, MySQL will accept all of the following strings as valid representations of 3:30 P.M. on September 17, 2019:

```
'2019-09-17 15:30:00'
'2019/09/17 15:30:00'
'2019,09,17,15,30,00'
'20190917153000'
```

Although this gives you a bit more flexibility, you may find yourself trying to generate a temporal value *without* the default date components; the next section demonstrates a built-in function that is far more flexible than the `cast()` function.

FUNCTIONS FOR GENERATING DATES

If you need to generate temporal data from a string, and the string is not in the proper form to use the `cast()` function, you can use a built-in function that allows you to provide a format string along with the date string. MySQL includes the `str_to_date()` function for this purpose. Say, for example, that you pull the string `'September 17, 2019'` from a file and need to use it to update a `date` column. Since the string is not in the required YYYY-MM-DD format, you can use `str_to_date()` instead of reformatting the string so that you can use the `cast()` function, as in:

```
UPDATE rental
SET return_date = STR_TO_DATE('September 17, 2019', '%M %d, %Y')
WHERE rental_id = 99999;
```

The second argument in the call to `str_to_date()` defines the format of the date string, with, in this case, a month name (`%M`), a numeric day (`%d`), and a four-digit numeric year (`%Y`). While there are over 30 recognized format components, Table 7-4 defines the dozen or so most commonly used components.

Table 7-4. Date format components

Format component	Description
`%M`	Month name (`January` to `December`)
`%m`	Month numeric (`01` to `12`)
`%d`	Day numeric (`01` to `31`)
`%j`	Day of year (`001` to `366`)
`%W`	Weekday name (`Sunday` to `Saturday`)
`%Y`	Year, four-digit numeric
`%y`	Year, two-digit numeric
`%H`	Hour (`00` to `23`)

Format component	Description
`%h`	Hour (`01` to `12`)
`%i`	Minutes (`00` to `59`)
`%s`	Seconds (`00` to `59`)
`%f`	Microseconds (`000000` to `999999`)
`%p`	A.M. or P.M.

The `str_to_date()` function returns a `datetime`, `date`, or `time` value depending on the contents of the format string. For example, if the format string includes only `%H`, `%i`, and `%s`, then a `time` value will be returned.

> **NOTE**
>
> Oracle Database users can use the `to_date()` function in the same manner as MySQL's `str_to_date()` function. SQL Server includes a `convert()` function that is not quite as flexible as MySQL and Oracle Database; rather than supplying a custom format string, your date string must conform to one of 21 predefined formats.

If you are trying to generate the *current* date/time, then you won't need to build a string, because the following built-in functions will access the system clock and return the current date and/or time as a string for you:

```
mysql> SELECT CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP();
+----------------+----------------+---------------------+
| CURRENT_DATE() | CURRENT_TIME() | CURRENT_TIMESTAMP() |
+----------------+----------------+---------------------+
| 2019-06-05     | 16:54:36       | 2019-06-05 16:54:36 |
+----------------+----------------+---------------------+
1 row in set (0.12 sec)
```

The values returned by these functions are in the default format for the temporal type being returned. Oracle Database includes `current_date()` and `current_timestamp()` but not `current_time()`, and SQL Server includes only the `current_timestamp()` function.

Manipulating Temporal Data

This section explores the built-in functions that take date arguments and return dates, strings, or numbers.

TEMPORAL FUNCTIONS THAT RETURN DATES

Many of the built-in temporal functions take one date as an argument and return another date. MySQL's `date_add()` function, for example, allows you to add any kind of interval (e.g., days, months, years) to a specified date to generate another date. Here's an example that demonstrates how to add five days to the current date:

```
mysql> SELECT DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY);
+------------------------------------------+
| DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY) |
+------------------------------------------+
| 2019-06-10                               |
+------------------------------------------+
1 row in set (0.06 sec)
```

The second argument is composed of three elements: the `interval` keyword, the desired quantity, and the type of interval. Table 7-5 shows some of the commonly used interval types.

Table 7-5. Common interval types

Interval name	Description
`Second`	Number of seconds
`Minute`	Number of minutes
`Hour`	Number of hours
`Day`	Number of days
`Month`	Number of months
`Year`	Number of years
`Minute_second`	Number of minutes and seconds, separated by “:”
`Hour_second`	Number of hours, minutes, and seconds, separated by “:”

Interval name	Description
`Year_month`	Number of years and months, separated by “-”

While the first six types listed in Table 7-5 are pretty straightforward, the last three types require a bit more explanation since they have multiple elements. For example, if you are told that a film was actually returned 3 hours, 27 minutes, and 11 seconds later than what was originally specified, you can fix it via the following:

```
UPDATE rental
SET return_date = DATE_ADD(return_date, INTERVAL '3:27:11' HOUR_SECOND)
WHERE rental_id = 99999;
```

In this example, the `date_add()` function takes the value in the `return_date` column, adds 3 hours, 27 minutes, and 11 seconds to it, and uses the value that results to modify the `return_date` column.

Or, if you work in HR and found out that employee ID 4789 claimed to be older than he actually is, you could add 9 years and 11 months to his birth date, as in:

```
UPDATE employee
SET birth_date = DATE_ADD(birth_date, INTERVAL '9-11' YEAR_MONTH)
WHERE emp_id = 4789;
```

> **NOTE**
>
> SQL Server users can accomplish the previous example using the `dateadd()` function:
>
> ```
> UPDATE employee
> SET birth_date =
>   DATEADD(MONTH, 119, birth_date)
> WHERE emp_id = 4789
> ```
>
> SQL Server doesn't have combined intervals (i.e., year_month), so I converted 9 years, 11 months to 119 months.
>
> Oracle Database users can use the add_months() function for this example, as in:
>
> ```
> UPDATE employee
> SET birth_date = ADD_MONTHS(birth_date, 119)
> WHERE emp_id = 4789;
> ```

There are some cases where you want to add an interval to a date, and you know where you want to arrive but not how many days it takes to get there. For example, let's say that a bank customer logs on to the online banking system and schedules a transfer for the end of the month. Rather than writing some code that figures out the current month and then looks up the number of days in that month, you can call the `last_day()` function, which does the work for you (both MySQL and Oracle Database include the `last_day()` function; SQL Server has no comparable function). If the customer asks for the transfer on September 17, 2019, you could find the last day of September via the following:

```
mysql> SELECT LAST_DAY('2019-09-17');
+------------------------+
| LAST_DAY('2019-09-17') |
+------------------------+
| 2019-09-30             |
+------------------------+
1 row in set (0.10 sec)
```

Whether you provide a `date` or `datetime` value, the `last_day()` function always returns a `date`. Although this function may not seem like an enormous timesaver, the underlying logic can be tricky if you're trying to find the last day of February and need to figure out whether the current year is a leap year.

TEMPORAL FUNCTIONS THAT RETURN STRINGS

Most of the temporal functions that return string values are used to extract a portion of a date or time. For example, MySQL includes the `dayname()` function to determine which day of the week a certain date falls on, as in:

```
mysql> SELECT DAYNAME('2019-09-18');
+-----------------------+
| DAYNAME('2019-09-18') |
+-----------------------+
| Wednesday             |
+-----------------------+
1 row in set (0.00 sec)
```

Many such functions are included with MySQL for extracting information from date values, but I recommend that you use the `extract()` function instead, since it's easier to remember a few variations of one function than to remember a dozen different functions. Additionally, the `extract()` function is part of the SQL:2003 standard and has been implemented by Oracle Database as well as MySQL.

The `extract()` function uses the same interval types as the `date_add()` function (see Table 7-5) to define which element of the date interests you. For example, if you want to extract just the year portion of a `datetime` value, you can do the following:

```
mysql> SELECT EXTRACT(YEAR FROM '2019-09-18 22:19:05');
+-------------------------------------------+
| EXTRACT(YEAR FROM '2019-09-18 22:19:05') |
+-------------------------------------------+
|                                      2019 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

NOTE

SQL Server doesn't include an implementation of `extract()`, but it does include the `datepart()` function. Here's how you would extract the year from a `datetime` value using `datepart()`:

```
SELECT DATEPART(YEAR, GETDATE())
```

TEMPORAL FUNCTIONS THAT RETURN NUMBERS

Earlier in this chapter, I showed you a function used to add a given interval to a date value, thus generating another date value. Another common activity when working with dates is to take *two* date values and determine the number of intervals (days, weeks, years) *between* the two dates. For this purpose, MySQL includes the function `datediff()`, which returns the number of full days between two dates. For example, if I want to know the number of days that my kids will be out of school this summer, I can do the following:

```
mysql> SELECT DATEDIFF('2019-09-03', '2019-06-21');
+--------------------------------------+
| DATEDIFF('2019-09-03', '2019-06-21') |
+--------------------------------------+
|                                   74 |
+--------------------------------------+
1 row in set (0.00 sec)
```

Thus, I will have to endure 74 days of poison ivy, mosquito bites, and scraped knees before the kids are safely back at school. The `datediff()` function ignores the time of day in its arguments. Even if I include a time-of-day, setting it to one second until midnight for the first date and to one second after midnight for the second date, those times will have no effect on the calculation:

```
mysql> SELECT DATEDIFF('2019-09-03 23:59:59', '2019-06-21 00:00:01');
+--------------------------------------------------------+
| DATEDIFF('2019-09-03 23:59:59', '2019-06-21 00:00:01') |
+--------------------------------------------------------+
|                                                     74 |
+--------------------------------------------------------+
1 row in set (0.00 sec)
```

If I switch the arguments and have the earlier date first, `datediff()` will return a negative number, as in:

```
mysql> SELECT DATEDIFF('2019-06-21', '2019-09-03');
+--------------------------------------+
| DATEDIFF('2019-06-21', '2019-09-03') |
+--------------------------------------+
|                                  -74 |
+--------------------------------------+
1 row in set (0.00 sec)
```

NOTE

SQL Server also includes the `datediff()` function, but it is more flexible than the MySQL implementation in that you can specify the interval type (i.e., year, month, day, hour) instead of counting only the number of days between two dates. Here's how SQL Server would accomplish the previous example:

```
SELECT DATEDIFF(DAY, '2019-06-21', '2019-09-03')
```

Oracle Database allows you to determine the number of days between two dates simply by subtracting one date from another.

Conversion Functions

Earlier in this chapter, I showed you how to use the `cast()` function to convert a string to a `datetime` value. While every database server includes a number of proprietary functions used to convert data from one type to another, I recommend using the `cast()` function, which is included in the SQL:2003 standard and has been implemented by MySQL, Oracle Database, and Microsoft SQL Server.

To use `cast()`, you provide a value or expression, the `as` keyword, and the type to which you want the value converted. Here's an example that converts a string to an integer:

```
mysql> SELECT CAST('1456328' AS SIGNED INTEGER);
+-----------------------------------+
| CAST('1456328' AS SIGNED INTEGER) |
+-----------------------------------+
|                           1456328 |
+-----------------------------------+
1 row in set (0.01 sec)
```

When converting a string to a number, the `cast()` function will attempt to convert the entire string from left to right; if any non-numeric characters are found in the string, the conversion halts without an error. Consider the following example:

```
mysql> SELECT CAST('999ABC111' AS UNSIGNED INTEGER);
+---------------------------------------+
| CAST('999ABC111' AS UNSIGNED INTEGER) |
+---------------------------------------+
|                                   999 |
+---------------------------------------+
1 row in set, 1 warning (0.08 sec)
```

```
mysql> show warnings;
+---------+------+------------------------------------------------+
| Level   | Code | Message                                        |
+---------+------+------------------------------------------------+
| Warning | 1292 | Truncated incorrect INTEGER value: '999ABC111' |
+---------+------+------------------------------------------------+
1 row in set (0.07 sec)
```

In this case, the first three digits of the string are converted, whereas the rest of the string is discarded, resulting in a value of `999`. The server did, however, issue a warning to let you know that not all the string was converted.

If you are converting a string to a `date`, `time`, or `datetime` value, then you will need to stick with the default formats for each type, since you can't provide the `cast()` function with a format string. If your date string is not in the default format (i.e., YYYY-MM-DD HH:MI:SS for `datetime` types), then you will need to resort to using another function, such as MySQL's `str_to_date()` function described earlier in the chapter.

Test Your Knowledge

These exercises are designed to test your understanding of some of the built-in functions shown in this chapter. See Appendix C for the answers.

Exercise 7-1

Write a query that returns the 17th through 25th characters of the string `'Please find the substring in this string'`.

Exercise 7-2

Write a query that returns the absolute value and sign (`−1`, `0`, or `1`) of the number `−25.76823`. Also return the number rounded to the nearest hundredth.

Exercise 7-3

Write a query to return just the month portion of the current date.

Chapter 8. Grouping and Aggregates

Data is generally stored at the lowest level of granularity needed by any of a database's users; if Chuck in accounting needs to look at individual customer transactions, then there needs to be a table in the database that stores individual transactions. That doesn't mean, however, that all users must deal with the data as it is stored in the database. The focus of this chapter is on how data can be grouped and aggregated to allow users to interact with it at some higher level of granularity than what is stored in the database.

Grouping Concepts

Sometimes you will want to find trends in your data that will require the database server to cook the data a bit before you can generate the results you are looking for. For example, let's say that you are in charge of sending coupons for free rentals to your best customers. You could issue a simple query to look at the raw data:

```
mysql> SELECT customer_id FROM rental;
+-------------+
| customer_id |
+-------------+
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
```

```
|           1 |
|           1 |
...
|         599 |
|         599 |
|         599 |
|         599 |
|         599 |
|         599 |
+-------------+
16044 rows in set (0.01 sec)
```

With 599 customers spanning over 16,000 rental records, it isn't feasible to determine which customers have rented the most films by looking at the raw data. Instead, you can ask the database server to group the data for you by using the `group by` clause. Here's the same query but employing a `group by` clause to group the rental data by customer ID:

```
mysql> SELECT customer_id
    -> FROM rental
    -> GROUP BY customer_id;
+-------------+
| customer_id |
+-------------+
|           1 |
|           2 |
|           3 |
|           4 |
|           5 |
|           6 |
...
|         594 |
|         595 |
|         596 |
|         597 |
|         598 |
|         599 |
+-------------+
599 rows in set (0.00 sec)
```

The result set contains one row for each distinct value in the `customer_id` column, resulting in 599 rows instead of the full 16,044 rows. The reason for the smaller result set is that some of the customers rented more than one film. To see how many films each customer opened, you can use an *aggregate function* in the `select` clause to count the number of rows in each group:

```
mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> GROUP BY customer_id;
+-------------+----------+
| customer_id | count(*) |
+-------------+----------+
|           1 |       32 |
|           2 |       27 |
|           3 |       26 |
|           4 |       22 |
|           5 |       38 |
|           6 |       28 |
...
|         594 |       27 |
|         595 |       30 |
|         596 |       28 |
|         597 |       25 |
|         598 |       22 |
|         599 |       19 |
+-------------+----------+
599 rows in set (0.01 sec)
```

The aggregate function `count()` counts the number of rows in each group, and the asterisk tells the server to count everything in the group. Using the combination of a `group by` clause and the `count()` aggregate function, you are able to generate exactly the data needed to answer the business question without having to look at the raw data.

Looking at the results, you can see that 32 films were rented by customer ID 1, and 25 films were rented by the customer ID 597. In order to determine which customers have rented the most films, simply add an`order by` clause:

```
mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> GROUP BY customer_id
    -> ORDER BY 2 DESC;
+-------------+----------+
| customer_id | count(*) |
+-------------+----------+
|         148 |       46 |
|         526 |       45 |
|         236 |       42 |
|         144 |       42 |
|          75 |       41 |
...
|         248 |       15 |
|         110 |       14 |
|         281 |       14 |
|          61 |       14 |
|         318 |       12 |
+-------------+----------+
599 rows in set (0.01 sec)
```

Now that the results are sorted, you can easily see that customer ID 148 has rented the most films (46), while customer ID 318 has rented the fewest films (12).

When grouping data, you may need to filter out undesired data from your result set based on groups of data rather than based on the raw data. Since the `group by` clause runs *after* the `where` clause has been evaluated, you cannot add filter conditions to your `where` clause for this purpose. For example, here’s an attempt to filter out any customers who have rented fewer than 40 films:

```
mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> WHERE count(*) >= 40
    -> GROUP BY customer_id;
ERROR 1111 (HY000): Invalid use of group function
```

You cannot refer to the aggregate function `count(*)` in your `where` clause, because the groups have not yet been generated at the time the `where` clause is evaluated. Instead, you must put your group filter conditions in the `having` clause. Here's what the query would look like using `having`:

```
mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> GROUP BY customer_id
    -> HAVING count(*) >= 40;
+-------------+----------+
| customer_id | count(*) |
+-------------+----------+
|          75 |       41 |
|         144 |       42 |
|         148 |       46 |
|         197 |       40 |
|         236 |       42 |
|         469 |       40 |
|         526 |       45 |
+-------------+----------+
7 rows in set (0.01 sec)
```

Because those groups containing fewer than 40 members have been filtered out via the `having` clause, the result set now contains only those customers who have rented 40 or more films.

Aggregate Functions

Aggregate functions perform a specific operation over all rows in a group. Although every database server has its own set of specialty aggregate functions, the common aggregate functions implemented by all major servers include:

`Max()`

Returns the maximum value within a set

`Min()`

Returns the minimum value within a set

`Avg()`

Returns the average value across a set

`Sum()`

Returns the sum of the values across a set

`Count()`

Returns the number of values in a set

Here's a query that uses all of the common aggregate functions to analyze the data on film rental payments:

```
mysql> SELECT MAX(amount) max_amt,
    ->   MIN(amount) min_amt,
    ->   AVG(amount) avg_amt,
    ->   SUM(amount) tot_amt,
    ->   COUNT(*) num_payments
    -> FROM payment;
+---------+---------+----------+----------+--------------+
| max_amt | min_amt | avg_amt  | tot_amt  | num_payments |
+---------+---------+----------+----------+--------------+
|   11.99 |    0.00 | 4.200667 | 67416.51 |        16049 |
+---------+---------+----------+----------+--------------+
1 row in set (0.09 sec)
```

The results from this query tell you that, across the 16,049 rows in the `payment` table, the maximum amount paid to rent a film was

$11.99, the minimum amount was $0, the average payment was $4.20, and the total of all rental payments was $67,416.51. Hopefully, this gives you an appreciation for the role of these aggregate functions; the next subsections further clarify how you can utilize these functions.

Implicit Versus Explicit Groups

In the previous example, every value returned by the query is generated by an aggregate function. Since there is no `group by` clause, there is a single, *implicit* group (all rows in the `payment` table).

In most cases, however, you will want to retrieve additional columns along with columns generated by aggregate functions. What if, for example, you wanted to extend the previous query to execute the same five aggregate functions for *each* customer, instead of across all customers? For this query, you would want to retrieve the `customer_id` column along with the five aggregate functions, as in:

```
SELECT customer_id,
  MAX(amount) max_amt,
  MIN(amount) min_amt,
  AVG(amount) avg_amt,
  SUM(amount) tot_amt,
  COUNT(*) num_payments
FROM payment;
```

However, if you try to execute the query, you will receive the following error:

```
ERROR 1140 (42000): In aggregated query without GROUP BY,
  expression #1 of SELECT list contains nonaggregated column
  'sakila.payment.customer_id';
  this is incompatible with sql_mode=only_full_group_by
```

While it may be obvious to you that you want the aggregate functions applied to each customer found in the `payment` table, this query fails because you have not *explicitly* specified how the data should be grouped. Therefore, you will need to add a `group by` clause to specify over which group of rows the aggregate functions should be applied:

```
mysql> SELECT customer_id,
    ->   MAX(amount) max_amt,
    ->   MIN(amount) min_amt,
    ->   AVG(amount) avg_amt,
    ->   SUM(amount) tot_amt,
    ->   COUNT(*) num_payments
    -> FROM payment
    -> GROUP BY customer_id;
+-------------+---------+---------+----------+---------+----
----------+
| customer_id | max_amt | min_amt | avg_amt  | tot_amt |
num_payments |
+-------------+---------+---------+----------+---------+----
----------+
|           1 |    9.99 |    0.99 | 3.708750 |  118.68
|           32 |
|           2 |   10.99 |    0.99 | 4.767778 |  128.73
|           27 |
|           3 |   10.99 |    0.99 | 5.220769 |  135.74
|           26 |
|           4 |    8.99 |    0.99 | 3.717273 |   81.78
|           22 |
|           5 |    9.99 |    0.99 | 3.805789 |  144.62
|           38 |
|           6 |    7.99 |    0.99 | 3.347143 |   93.72
|           28 |
...
|         594 |    8.99 |    0.99 | 4.841852 |  130.73
|           27 |
|         595 |   10.99 |    0.99 | 3.923333 |  117.70
|           30 |
```

```
|          596 |    6.99 |    0.99 | 3.454286 |   96.72
|            28 |
|          597 |    8.99 |    0.99 | 3.990000 |   99.75
|            25 |
|          598 |    7.99 |    0.99 | 3.808182 |   83.78
|            22 |
|          599 |    9.99 |    0.99 | 4.411053 |   83.81
|            19 |
+-------------+---------+---------+----------+---------+----
----------+
599 rows in set (0.04 sec)
```

With the inclusion of the `group by` clause, the server knows to group together rows having the same value in the `customer_id` column first and then to apply the five aggregate functions to each of the 599 groups.

Counting Distinct Values

When using the `count()` function to determine the number of members in each group, you have your choice of counting *all* members in the group, or counting only the *distinct* values for a column across all members of the group. For example, consider the following query, which uses the `count()` function with the `customer_id` column in two different ways:

```
mysql> SELECT COUNT(customer_id) num_rows,
    ->   COUNT(DISTINCT customer_id) num_customers
    -> FROM payment;
+----------+---------------+
| num_rows | num_customers |
+----------+---------------+
|    16049 |           599 |
+----------+---------------+
1 row in set (0.01 sec)
```

The first column in the query simply counts the number of rows in the `payment` table, whereas the second column examines the values in the `customer_id` column and counts only the number of unique values. By specifying `distinct`, therefore, the `count()` function examines the values of a column for each member of the group in order to find and remove duplicates, rather than simply counting the number of values in the group.

Using Expressions

Along with using columns as arguments to aggregate functions, you can use expressions as well. For example, you may want to find the maximum number of days between when a film was rented and subsequently returned. You can achieve this via the following query:

```
mysql> SELECT MAX(datediff(return_date,rental_date))
    -> FROM rental;
+----------------------------------------+
| MAX(datediff(return_date,rental_date)) |
+----------------------------------------+
|                                     33 |
+----------------------------------------+
1 row in set (0.01 sec)
```

The `datediff` function is used to compute the number of days between the return date and the rental date for every rental, and the `max` function returns the highest value, which in this case is 33 days.

While this example uses a fairly simple expression, expressions used as arguments to aggregate functions can be as complex as needed, as long as they return a number, string, or date. In Chapter 11, I show you how you can use `case` expressions with aggregate functions to determine whether a particular row should or should not be included in an aggregation.

How Nulls Are Handled

When performing aggregations, or, indeed, any type of numeric calculation, you should always consider how `null` values might affect the outcome of your calculation. To illustrate, I will build a simple table to hold numeric data and populate it with the set {1, 3, 5}:

```
mysql> CREATE TABLE number_tbl
    ->  (val SMALLINT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO number_tbl VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

Consider the following query, which performs five aggregate functions on the set of numbers:

```
mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        3 |        3 |     9 |       5 |  3.0000 |
+----------+----------+-------+---------+---------+
1 row in set (0.08 sec)
```

The results are as you would expect: both `count(*)` and `count(val)` return the value 3, `sum(val)` returns the value 9, `max(val)` returns 5, and `avg(val)` returns 3. Next, I will add a `null` value to the `number_tbl` table and run the query again:

```
mysql> INSERT INTO number_tbl VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        4 |        3 |     9 |       5 |  3.0000 |
+----------+----------+-------+---------+---------+
1 row in set (0.00 sec)
```

Even with the addition of the `null` value to the table, the `sum()`, `max()`, and `avg()` functions all return the same values, indicating that they ignore any `null` values encountered. The `count(*)` function now returns the value 4, which is valid since the `number_tbl` table contains four rows, while the `count(val)` function still returns the value 3. The difference is that `count(*)` counts the number of rows, whereas `count(val)` counts the number of *values* contained in the `val` column and ignores any `null` values encountered.

Generating Groups

People are rarely interested in looking at raw data; instead, people engaging in data analysis will want to manipulate the raw data to better suit their needs. Examples of common data manipulations include:

- Generating totals for a geographic region, such as total European sales
- Finding outliers, such as the top salesperson for 2020
- Determining frequencies, such as the number of films rented in each month

To answer these types of queries, you will need to ask the database server to group rows together by one or more columns or expressions. As you have seen already in several examples, the `group by` clause is the mechanism for grouping data within a query. In this section, you will see how to group data by one or more columns, how to group data using expressions, and how to generate rollups within groups.

Single-Column Grouping

Single-column groups are the simplest and most-often-used type of grouping. If you want to find the number of films associated with each actor, for example, you need only group on the `film_actor.actor_id` column, as in:

```
mysql> SELECT actor_id, count(*)
    -> FROM film_actor
    -> GROUP BY actor_id;
+----------+----------+
| actor_id | count(*) |
+----------+----------+
```

```
|        1 |       19 |
|        2 |       25 |
|        3 |       22 |
|        4 |       22 |
...
|      197 |       33 |
|      198 |       40 |
|      199 |       15 |
|      200 |       20 |
+----------+----------+
200 rows in set (0.11 sec)
```

This query generates 200 groups, one for each actor, and then sums the number of films for each member of the group.

Multicolumn Grouping

In some cases, you may want to generate groups that span *more* than one column. Expanding on the previous example, imagine that you want to find the total number of films for each film rating (G, PG, …) for each actor. The following example shows how you can accomplish this:

```
mysql> SELECT fa.actor_id, f.rating, count(*)
    -> FROM film_actor fa
    ->   INNER JOIN film f
    ->   ON fa.film_id = f.film_id
    -> GROUP BY fa.actor_id, f.rating
    -> ORDER BY 1,2;
+----------+--------+----------+
| actor_id | rating | count(*) |
+----------+--------+----------+
|        1 | G      |        4 |
|        1 | PG     |        6 |
|        1 | PG-13  |        1 |
|        1 | R      |        3 |
|        1 | NC-17  |        5 |
|        2 | G      |        7 |
|        2 | PG     |        6 |
|        2 | PG-13  |        2 |
|        2 | R      |        2 |
```

```
|        2 | NC-17  |        8 |
...
|      199 | G      |        3 |
|      199 | PG     |        4 |
|      199 | PG-13  |        4 |
|      199 | R      |        2 |
|      199 | NC-17  |        2 |
|      200 | G      |        5 |
|      200 | PG     |        3 |
|      200 | PG-13  |        2 |
|      200 | R      |        6 |
|      200 | NC-17  |        4 |
+----------+--------+----------+
996 rows in set (0.01 sec)
```

This version of the query generates 996 groups, one for each combination of actor and film rating found by joining the `film_actor` table with the `film` table. Along with adding the `rating` column to the `select` clause, I also added it to the `group by` clause, since `rating` is retrieved from a table and is not generated via an aggregate function such as **max** or **count**.

Grouping via Expressions

Along with using columns to group data, you can build groups based on the values generated by expressions. Consider the following query, which groups rentals by year:

```
mysql> SELECT extract(YEAR FROM rental_date) year,
    ->   COUNT(*) how_many
    -> FROM rental
    -> GROUP BY extract(YEAR FROM rental_date);
+------+----------+
| year | how_many |
+------+----------+
| 2005 |    15862 |
| 2006 |      182 |
+------+----------+
2 rows in set (0.01 sec)
```

This query employs a fairly simple expression, which uses the `extract()` function to return only the year portion of a date, to group the rows in the `rental` table.

Generating Rollups

In "Multicolumn Grouping", I showed an example that counts the number films for each actor and film rating. Let's say, however, that along with the total count for each actor/rating combination, you also want total counts for each distinct actor. You could run an additional query and merge the results, you could load the results of the query into a spreadsheet, or you could build a Python script, Java program, or some other mechanism to take that data and perform the additional calculations. Better yet, you could use the `with rollup` option to have the database server do the work for you. Here's the revised query using `with rollup` in the `group by` clause:

```
mysql> SELECT fa.actor_id, f.rating, count(*)
    -> FROM film_actor fa
    ->   INNER JOIN film f
    ->   ON fa.film_id = f.film_id
    -> GROUP BY fa.actor_id, f.rating WITH ROLLUP
    -> ORDER BY 1,2;
+----------+--------+----------+
| actor_id | rating | count(*) |
+----------+--------+----------+
|     NULL | NULL   |     5462 |
|        1 | NULL   |       19 |
|        1 | G      |        4 |
|        1 | PG     |        6 |
|        1 | PG-13  |        1 |
|        1 | R      |        3 |
|        1 | NC-17  |        5 |
|        2 | NULL   |       25 |
|        2 | G      |        7 |
|        2 | PG     |        6 |
|        2 | PG-13  |        2 |
```

```
|          2 | R        |          2 |
|          2 | NC-17    |          8 |
...
|        199 | NULL     |         15 |
|        199 | G        |          3 |
|        199 | PG       |          4 |
|        199 | PG-13    |          4 |
|        199 | R        |          2 |
|        199 | NC-17    |          2 |
|        200 | NULL     |         20 |
|        200 | G        |          5 |
|        200 | PG       |          3 |
|        200 | PG-13    |          2 |
|        200 | R        |          6 |
|        200 | NC-17    |          4 |
+----------+--------+----------+
1197 rows in set (0.07 sec)
```

There are now 201 additional rows in the result set, one for each of the 200 distinct actors and one for the grand total (all actors combined). For the 200 actor rollups, a `null` value is provided for the `rating` column, since the rollup is being performed across all ratings. Looking at the first line for actor_id 200, for example, you will see that a total of 20 films are associated with the actor; this equals the sum of the counts for each rating (4 NC-17 + 6 R + 2 PG-13 + 3 PG + 5 G). For the grand total row in the first line of the output, a `null` value is provided for both the `actor_id` and `rating` columns; the total for the first line of output equals 5,462, which is equal to the number of rows in the `film_actor` table.

> **NOTE**
>
> If you are using Oracle Database, you need to use a slightly different syntax to indicate that you want a rollup performed. The `group by` clause for the previous query would look as follows when using Oracle:
>
> ```
> GROUP BY ROLLUP(fa.actor_id, f.rating)
> ```
>
> The advantage of this syntax is that it allows you to perform rollups on a subset of the columns in the `group_by` clause. If you are grouping by columns a, b, and c, for example, you could indicate that the server should perform rollups on only b and c via the following:
>
> ```
> GROUP BY a, ROLLUP(b, c)
> ```

If, along with totals by actor, you also want to calculate totals per rating, then you can use the `with cube` option, which generates summary rows for *all* possible combinations of the grouping columns. Unfortunately, `with cube` is not available in version 8.0 of MySQL, but it is available with SQL Server and Oracle Database.

Group Filter Conditions

In Chapter 4, I introduced you to various types of filter conditions and showed how you can use them in the `where` clause. When grouping data, you also can apply filter conditions to the data *after* the groups have been generated. The `having` clause is where you should place these types of filter conditions. Consider the following example:

```
mysql> SELECT fa.actor_id, f.rating, count(*)
    -> FROM film_actor fa
    ->   INNER JOIN film f
    ->   ON fa.film_id = f.film_id
    -> WHERE f.rating IN ('G','PG')
    -> GROUP BY fa.actor_id, f.rating
    -> HAVING count(*) > 9;
+----------+--------+----------+
| actor_id | rating | count(*) |
+----------+--------+----------+
|      137 | PG     |       10 |
|       37 | PG     |       12 |
|      180 | PG     |       12 |
|        7 | G      |       10 |
|       83 | G      |       14 |
|      129 | G      |       12 |
|      111 | PG     |       15 |
|       44 | PG     |       12 |
|       26 | PG     |       11 |
|       92 | PG     |       12 |
|       17 | G      |       12 |
|      158 | PG     |       10 |
|      147 | PG     |       10 |
|       14 | G      |       10 |
|      102 | PG     |       11 |
|      133 | PG     |       10 |
+----------+--------+----------+
16 rows in set (0.01 sec)
```

This query has two filter conditions: one in the `where` clause, which filters out any films rated something other than G or PG, and another in the `having` clause, which filters out any actors who appeared in less than 10 films. Thus, one of the filters acts on data *before* it is grouped, and the other filter acts on data *after* the groups have been created. If you mistakenly put both filters in the `where` clause, you will see the following error:

```
mysql> SELECT fa.actor_id, f.rating, count(*)
    -> FROM film_actor fa
    ->   INNER JOIN film f
    ->   ON fa.film_id = f.film_id
    -> WHERE f.rating IN ('G','PG')
```

```
    ->   AND count(*) > 9
    -> GROUP BY fa.actor_id, f.rating;
ERROR 1111 (HY000): Invalid use of group function
```

This query fails because you cannot include an aggregate function in a query's `where` clause. This is because the filters in the `where` clause are evaluated *before* the grouping occurs, so the server can't yet perform any functions on groups.

> **WARNING**
>
> When adding filters to a query that includes a `group by` clause, think carefully about whether the filter acts on raw data, in which case it belongs in the `where` clause, or on grouped data, in which case it belongs in the `having` clause.

Test Your Knowledge

Work through the following exercises to test your grasp of SQL's grouping and aggregating features. Check your work with the answers in Appendix C.

Exercise 8-1

Construct a query that counts the number of rows in the `payment` table.

Exercise 8-2

Modify your query from Exercise 8-1 to count the number of payments made by each customer. Show the customer ID and the total amount paid for each customer.

Exercise 8-3

Modify your query from Exercise 8-2 to include only those customers having made at least five payments.

Chapter 9. Subqueries

Subqueries are a powerful tool that you can use in all four SQL data statements. In this chapter, I'll show you how subqueries can be used to filter data, generate values, and construct temporary data sets. After a little experimentation, I think you'll agree that subqueries are one of the most powerful features of the SQL language.

What Is a Subquery?

A *subquery* is a query contained within another SQL statement (which I refer to as the *containing statement* for the rest of this discussion). A subquery is always enclosed within parentheses, and it is usually executed prior to the containing statement. Like any query, a subquery returns a result set that may consist of:

- A single row with a single column
- Multiple rows with a single column
- Multiple rows having multiple columns

The type of result set returned by the subquery determines how it may be used and which operators the containing statement may use to interact with the data the subquery returns. When the containing statement has finished executing, the data returned by any subqueries is discarded, making a subquery act like a temporary table with *statement scope* (meaning that the server frees up any memory allocated to the subquery results after the SQL statement has finished execution).

You already saw several examples of subqueries in earlier chapters, but here's a simple example to get started:

```
mysql> SELECT customer_id, first_name, last_name
    -> FROM customer
    -> WHERE customer_id = (SELECT MAX(customer_id) FROM
customer);
+-------------+------------+-----------+
| customer_id | first_name | last_name |
+-------------+------------+-----------+
|         599 | AUSTIN     | CINTRON   |
+-------------+------------+-----------+
1 row in set (0.27 sec)
```

In this example, the subquery returns the maximum value found in the `customer_id` column in the `customer` table, and the containing statement then returns data about that customer. If you are ever confused about what a subquery is doing, you can run the subquery by itself (without the parentheses) to see what it returns. Here's the subquery from the previous example:

```
mysql> SELECT MAX(customer_id) FROM customer;
+------------------+
| MAX(customer_id) |
+------------------+
|              599 |
+------------------+
1 row in set (0.00 sec)
```

The subquery returns a single row with a single column, which allows it to be used as one of the expressions in an equality condition (if the subquery returned two or more rows, it could be *compared* to something but could not be *equal* to anything, but more on this later).

In this case, you can take the value the subquery returned and substitute it into the righthand expression of the filter condition in the containing query, as in:

```
mysql> SELECT customer_id, first_name, last_name
    -> FROM customer
    -> WHERE customer_id = 599;
+-------------+------------+-----------+
| customer_id | first_name | last_name |
+-------------+------------+-----------+
|         599 | AUSTIN     | CINTRON   |
+-------------+------------+-----------+
1 row in set (0.00 sec)
```

The subquery is useful in this case because it allows you to retrieve information about the customer with the highest ID in a single query, rather than retrieving the maximum `customer_id` using one query and then writing a second query to retrieve the desired data from the `customer` table. As you will see, subqueries are useful in many other situations as well, and may become one of the most powerful tools in your SQL toolkit.

Subquery Types

Along with the differences noted previously regarding the type of result set returned by a subquery (single row/column, single row/multicolumn, or multiple columns), you can use another feature to differentiate subqueries; some subqueries are completely self-contained (called *noncorrelated subqueries*), while others reference columns from the containing statement (called *correlated subqueries*). The next several sections explore these two subquery types and show the different operators that you can employ to interact with them.

Noncorrelated Subqueries

The example from earlier in the chapter is a noncorrelated subquery; it may be executed alone and does not reference anything from the containing statement. Most subqueries that you encounter will be of this type unless you are writing `update` or `delete` statements, which frequently make use of correlated subqueries (more on this later). Along with being noncorrelated, the example from earlier in the chapter also returns a result set containing a single row and column. This type of subquery is known as a *scalar subquery* and can appear on either side of a condition using the usual operators (=, <>, <, >, <=, >=). The next example shows how you can use a scalar subquery in an inequality condition:

```
mysql> SELECT city_id, city
    -> FROM city
    -> WHERE country_id <>
    ->  (SELECT country_id FROM country WHERE country =
'India');
+---------+----------------------------+
| city_id | city                       |
+---------+----------------------------+
|       1 | A Corua (La Corua)         |
|       2 | Abha                       |
|       3 | Abu Dhabi                  |
|       4 | Acua                       |
|       5 | Adana                      |
|       6 | Addis Abeba                |
...
|     595 | Zapopan                    |
|     596 | Zaria                      |
|     597 | Zeleznogorsk               |
|     598 | Zhezqazghan                |
```

```
|     599 | Zhoushan                   |
|     600 | Ziguinchor                 |
+---------+----------------------------+
540 rows in set (0.02 sec)
```

This query returns all cities which are not in India. The subquery, which is found on the last line of the statement, returns the country ID for India, and the containing query returns all cities which do not have that country ID. While the subquery in this example is quite simple, subqueries may be as complex as you need them to be, and they may utilize any and all the available query clauses (`select`, `from`, `where`, `group by`, `having`, and `order by`).

If you use a subquery in an equality condition, but the subquery returns more than one row, you will receive an error. For example, if you modify the previous query such that the subquery returns *all* countries *except for* India, you will receive the following error:

```
mysql> SELECT city_id, city
    -> FROM city
    -> WHERE country_id <>
    ->  (SELECT country_id FROM country WHERE country <>
'India');
ERROR 1242 (21000): Subquery returns more than 1 row
```

If you run the subquery by itself, you will see the following results:

```
mysql> SELECT country_id FROM country WHERE country <>
'India';
+------------+
| country_id |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
...
```

```
|        106 |
|        107 |
|        108 |
|        109 |
+------------+
108 rows in set (0.00 sec)
```

The containing query fails because an expression (`country_id`) cannot be equated to a set of expressions (`country_id`s 1, 2, 3, …, 109). In other words, a single thing cannot be equated to a set of things. In the next section, you will see how to fix the problem by using a different operator.

Multiple-Row, Single-Column Subqueries

If your subquery returns more than one row, you will not be able to use it on one side of an equality condition, as the previous example demonstrated. However, there are four additional operators that you can use to build conditions with these types of subqueries.

THE IN AND NOT IN OPERATORS

While you can’t *equate* a single value to a set of values, you can check to see whether a single value can be found *within* a set of values. The next example, while it doesn’t use a subquery, demonstrates how to build a condition that uses the `in` operator to search for a value within a set of values:

```
mysql> SELECT country_id
    -> FROM country
    -> WHERE country IN ('Canada','Mexico');
+------------+
| country_id |
+------------+
|         20 |
|         60 |
```

```
+------------+
2 rows in set (0.00 sec)
```

The expression on the lefthand side of the condition is the `country` column, while the righthand side of the condition is a set of strings. The `in` operator checks to see whether either of the strings can be found in the `country` column; if so, the condition is met and the row is added to the result set. You could achieve the same results using two equality conditions, as in:

```
mysql> SELECT country_id
    -> FROM country
    -> WHERE country = 'Canada' OR country = 'Mexico';
+------------+
| country_id |
+------------+
|         20 |
|         60 |
+------------+
2 rows in set (0.00 sec)
```

While this approach seems reasonable when the set contains only two expressions, it is easy to see why a single condition using the `in` operator would be preferable if the set contained dozens (or hundreds, thousands, etc.) of values.

Although you will occasionally create a set of strings, dates, or numbers to use on one side of a condition, you are more likely to generate the set using a subquery that returns one or more rows. The following query uses the `in` operator with a subquery on the righthand side of the filter condition to return all cities which are in Canada or Mexico:

```
mysql> SELECT city_id, city
    -> FROM city
    -> WHERE country_id IN
    ->  (SELECT country_id
    ->   FROM country
    ->   WHERE country IN ('Canada','Mexico'));
+---------+----------------------------+
| city_id | city                       |
+---------+----------------------------+
|     179 | Gatineau                   |
|     196 | Halifax                    |
|     300 | Lethbridge                 |
|     313 | London                     |
|     383 | Oshawa                     |
|     430 | Richmond Hill              |
|     565 | Vancouver                  |
...
|     452 | San Juan Bautista Tuxtepec |
|     541 | Torren                     |
|     556 | Uruapan                    |
|     563 | Valle de Santiago          |
|     595 | Zapopan                    |
+---------+----------------------------+
37 rows in set (0.00 sec)
```

Along with seeing whether a value exists within a set of values, you can check the converse using the `not in` operator. Here's another version of the previous query using `not in` instead of `in`:

```
mysql> SELECT city_id, city
    -> FROM city
    -> WHERE country_id NOT IN
    ->  (SELECT country_id
    ->   FROM country
    ->   WHERE country IN ('Canada','Mexico'));
+---------+----------------------------+
| city_id | city                       |
+---------+----------------------------+
|       1 | A Corua (La Corua)         |
|       2 | Abha                       |
|       3 | Abu Dhabi                  |
|       5 | Adana                      |
|       6 | Addis Abeba                |
...
|     596 | Zaria                      |
```

```
|     597 | Zeleznogorsk                |
|     598 | Zhezqazghan                 |
|     599 | Zhoushan                    |
|     600 | Ziguinchor                  |
+---------+-----------------------------+
563 rows in set (0.00 sec)
```

This query finds all cities which are *not* in Canada or Mexico.

THE ALL OPERATOR

While the `in` operator is used to see whether an expression can be found within a set of expressions, the `all` operator allows you to make comparisons between a single value and every value in a set. To build such a condition, you will need to use one of the comparison operators (=, <>, <, >, etc.) in conjunction with the `all` operator. For example, the next query finds all customers who have never gotten a free film rental:

```
mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE customer_id <> ALL
    ->  (SELECT customer_id
    ->   FROM payment
    ->   WHERE amount = 0);
+-------------+--------------+
| first_name  | last_name    |
+-------------+--------------+
| MARY        | SMITH        |
| PATRICIA    | JOHNSON      |
| LINDA       | WILLIAMS     |
| BARBARA     | JONES        |
...
| EDUARDO     | HIATT        |
| TERRENCE    | GUNDERSON    |
| ENRIQUE     | FORSYTHE     |
| FREDDIE     | DUGGAN       |
| WADE        | DELVALLE     |
| AUSTIN      | CINTRON      |
+-------------+--------------+
576 rows in set (0.01 sec)
```

The subquery returns the set of IDs for customers who have paid $0 for a film rental, and the containing query returns the names of all customers whose ID is not in the set returned by the subquery. If this approach seems a bit clumsy to you, you are in good company; most people would prefer to phrase the query differently and avoid using the `all` operator. To illustrate, the previous query generates the same results as the next example, which uses the `not in` operator:

```
SELECT first_name, last_name
FROM customer
WHERE customer_id NOT IN
 (SELECT customer_id
  FROM payment
  WHERE amount = 0)
```

It's a matter of preference, but I think that most people would find the version that uses `not in` to be easier to understand.

> **NOTE**
>
> When using `not in` or `<> all` to compare a value to a set of values, you must be careful to ensure that the set of values does not contain a `null` value, because the server equates the value on the lefthand side of the expression to each member of the set, and any attempt to equate a value to `null` yields `unknown`. Thus, the following query returns an empty set:
>
> ```
> mysql> SELECT first_name, last_name
>     -> FROM customer
>     -> WHERE customer_id NOT IN (122, 452, NULL);
> Empty set (0.00 sec)
> ```

Here's another example using the `all` operator, but this time the subquery is in the `having` clause:

```
mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> GROUP BY customer_id
    -> HAVING count(*) > ALL
    ->  (SELECT count(*)
    ->   FROM rental r
    ->     INNER JOIN customer c
    ->     ON r.customer_id = c.customer_id
    ->     INNER JOIN address a
    ->     ON c.address_id = a.address_id
    ->     INNER JOIN city ct
    ->     ON a.city_id = ct.city_id
    ->     INNER JOIN country co
    ->     ON ct.country_id = co.country_id
    ->   WHERE co.country IN ('United
States','Mexico','Canada')
    ->   GROUP BY r.customer_id
    ->   );
+-------------+----------+
| customer_id | count(*) |
+-------------+----------+
|         148 |       46 |
+-------------+----------+
1 row in set (0.01 sec)
```

The subquery in this example returns the total number of film rentals for all customers in North America, and the containing query returns all customers whose total number of film rentals exceeds any of the North American customers.

THE ANY OPERATOR

Like the `all` operator, the `any` operator allows a value to be compared to the members of a set of values; unlike `all`, however, a condition using the `any` operator evaluates to `true` as soon as a single comparison is favorable. This example will find all customers whose total film rental payments exceed the total payments for all customers in Bolivia, Paraguay, or Chile:

```
mysql> SELECT customer_id, sum(amount)
    -> FROM payment
    -> GROUP BY customer_id
    -> HAVING sum(amount) > ANY
    ->  (SELECT sum(p.amount)
    ->   FROM payment p
    ->     INNER JOIN customer c
    ->     ON p.customer_id = c.customer_id
    ->     INNER JOIN address a
    ->     ON c.address_id = a.address_id
    ->     INNER JOIN city ct
    ->     ON a.city_id = ct.city_id
    ->     INNER JOIN country co
    ->     ON ct.country_id = co.country_id
    ->   WHERE co.country IN ('Bolivia','Paraguay','Chile')
    ->   GROUP BY co.country
    ->  );
+-------------+-------------+
| customer_id | sum(amount) |
+-------------+-------------+
|         137 |      194.61 |
|         144 |      195.58 |
|         148 |      216.54 |
|         178 |      194.61 |
|         459 |      186.62 |
|         526 |      221.55 |
+-------------+-------------+
6 rows in set (0.03 sec)
```

The subquery returns the total film rental fees for all customers in Bolivia, Paraguay, and Chile, and the containing query returns all customers who outspent at least one of these 3 countries (if you find yourself outspending an entire country, perhaps you need to cancel your Netflix subscription and book a trip to Bolivia, Paraguay, or Chile...).

> **NOTE**
>
> Although most people prefer to use `in`, using `= any` is equivalent to using the `in` operator.

Multicolumn Subqueries

So far, all of the subquery examples in this chapter have returned a single column and one or more rows. In certain situations, however, you can use subqueries that return two or more columns. To show the utility of multiple-column subqueries, it might help to look first at an example that uses multiple, single-column subqueries:

```
mysql> SELECT fa.actor_id, fa.film_id
    -> FROM film_actor fa
    -> WHERE fa.actor_id IN
    ->  (SELECT actor_id FROM actor WHERE last_name =
'MONROE')
    ->   AND fa.film_id IN
    ->  (SELECT film_id FROM film WHERE rating = 'PG');
+----------+---------+
| actor_id | film_id |
+----------+---------+
|      120 |      63 |
|      120 |     144 |
|      120 |     414 |
|      120 |     590 |
|      120 |     715 |
|      120 |     894 |
|      178 |     164 |
|      178 |     194 |
|      178 |     273 |
|      178 |     311 |
|      178 |     983 |
+----------+---------+
11 rows in set (0.00 sec)
```

This query uses two subqueries to identify all actors with the last name Monroe and all films rated PG, and the containing query then uses this information to retrieve all cases where an actor named Monroe appeared in a PG film. However, you could merge the two single-column subqueries into one multi-column subquery, and compare the results to two columns in the `film_actor` table. To do so, your filter condition must name both columns from the `film_actor` table surrounded by parentheses and in the same order as returned by the subquery, as in:

```
mysql> SELECT actor_id, film_id
    -> FROM film_actor
    -> WHERE (actor_id, film_id) IN
    ->  (SELECT a.actor_id, f.film_id
    ->   FROM actor a
    ->      CROSS JOIN film f
    ->   WHERE a.last_name = 'MONROE'
    ->   AND f.rating = 'PG');
+----------+---------+
| actor_id | film_id |
+----------+---------+
|      120 |      63 |
|      120 |     144 |
|      120 |     414 |
|      120 |     590 |
|      120 |     715 |
|      120 |     894 |
|      178 |     164 |
|      178 |     194 |
|      178 |     273 |
|      178 |     311 |
|      178 |     983 |
+----------+---------+
11 rows in set (0.00 sec)
```

This version of the query performs the same function as the previous example, but with a single subquery that returns two columns instead of two subqueries that each return a single column. The subquery in this version uses a type of join called a **Cross Join**, which will be explored in the next chapter, but the basic ideas is to return all combinations of actors named Monroe (2) and all films rated PG (194) for a total of 388 rows, 11 of which can be found in the `film_actor` table.

Correlated Subqueries

All of the subqueries shown thus far have been independent of their containing statements, meaning that you can execute them by themselves and inspect the results. A *correlated subquery*, on the other hand, is *dependent* on its containing statement from which it references one or more columns. Unlike a noncorrelated subquery, a correlated subquery is not executed once prior to execution of the containing statement; instead, the correlated subquery is executed once for each candidate row (rows that might be included in the final results). For example, the following query uses a correlated subquery to count the number of film rentals for each customer, and the containing query then retrieves those customers having rented exactly twenty films:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE 20 =
    ->  (SELECT count(*) FROM rental r
    ->   WHERE r.customer_id = c.customer_id);
+------------+-------------+
| first_name | last_name   |
+------------+-------------+
| LAUREN     | HUDSON      |
| JEANETTE   | GREENE      |
| TARA       | RYAN        |
| WILMA      | RICHARDS    |
| JO         | FOWLER      |
```

```
| KAY        | CALDWELL    |
| DANIEL     | CABRAL      |
| ANTHONY    | SCHWAB      |
| TERRY      | GRISSOM     |
| LUIS       | YANEZ       |
| HERBERT    | KRUGER      |
| OSCAR      | AQUINO      |
| RAUL       | FORTIER     |
| NELSON     | CHRISTENSON |
| ALFREDO    | MCADAMS     |
+------------+-------------+
15 rows in set (0.01 sec)
```

The reference to `c.customer_id` at the very end of the subquery is what makes the subquery correlated; the containing query must supply values for `c.customer_id` for the subquery to execute. In this case, the containing query retrieves all 599 rows from the `customer` table and executes the subquery once for each customer, passing in the appropriate customer ID for each execution. If the subquery returns the value `20`, then the filter condition is met and the row is added to the result set.

NOTE

One word of caution: since the correlated subquery will be executed once for each row of the containing query, the use of correlated subqueries can cause performance issues if the containing query returns a large number of rows.

Along with equality conditions, you can use correlated subqueries in other types of conditions, such as the range condition illustrated here:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE
    ->  (SELECT sum(p.amount) FROM payment p
```

```
    ->   WHERE p.customer_id = c.customer_id)
    ->   BETWEEN 180 AND 240;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| RHONDA     | KENNEDY   |
| CLARA      | SHAW      |
| ELEANOR    | HUNT      |
| MARION     | SNYDER    |
| TOMMY      | COLLAZO   |
| KARL       | SEAL      |
+------------+-----------+
6 rows in set (0.03 sec)
```

This variation on the previous query finds all customers whose total payments for all film rentals lies between $180 and $240. Once again, the correlated subquery is executed 599 times (once for each customer row), and each execution of the subquery returns the total account balance for the given customer.

> **NOTE**
>
> Another subtle difference in the previous query is that the subquery is on the lefthand side of the condition, which may look a bit odd but is perfectly valid.

The exists Operator

While you will often see correlated subqueries used in equality and range conditions, the most common operator used to build conditions that utilize correlated subqueries is the `exists` operator. You use the `exists` operator when you want to identify that a relationship exists without regard for the quantity; for example, the following query finds all the customers who rented at least one film prior to May 25, 2005, without regard for how many films were rented:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE EXISTS
    ->  (SELECT 1 FROM rental r
    ->   WHERE r.customer_id = c.customer_id
    ->     AND date(r.rental_date) < '2005-05-25');
+------------+-------------+
| first_name | last_name   |
+------------+-------------+
| CHARLOTTE  | HUNTER      |
| DELORES    | HANSEN      |
| MINNIE     | ROMERO      |
| CASSANDRA  | WALTERS     |
| ANDREW     | PURDY       |
| MANUEL     | MURRELL     |
| TOMMY      | COLLAZO     |
| NELSON     | CHRISTENSON |
+------------+-------------+
8 rows in set (0.03 sec)
```

Using the `exists` operator, your subquery can return zero, one, or many rows, and the condition simply checks whether the subquery returned one or more rows. If you look at the `select` clause of the subquery, you will see that it consists of a single literal (`1`); since the condition in the containing query only needs to know how many rows have been returned, the actual data the subquery returned is irrelevant. Your subquery can return whatever strikes your fancy, as demonstrated next:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE EXISTS
    ->  (SELECT r.rental_date, r.customer_id, 'ABCD' str, 2
* 3 / 7 nmbr
    ->   FROM rental r
    ->   WHERE r.customer_id = c.customer_id
    ->     AND date(r.rental_date) < '2005-05-25');
+------------+-------------+
| first_name | last_name   |
+------------+-------------+
| CHARLOTTE  | HUNTER      |
```

```
| DELORES    | HANSEN      |
| MINNIE     | ROMERO      |
| CASSANDRA  | WALTERS     |
| ANDREW     | PURDY       |
| MANUEL     | MURRELL     |
| TOMMY      | COLLAZO     |
| NELSON     | CHRISTENSON |
+------------+-------------+
8 rows in set (0.03 sec)
```

However, the convention is to specify either `select 1` or `select *` when using `exists`.

You may also use `not exists` to check for subqueries that return no rows, as demonstrated by the following:

```
mysql> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE NOT EXISTS
    ->  (SELECT 1
    ->   FROM film_actor fa
    ->     INNER JOIN film f ON f.film_id = fa.film_id
    ->   WHERE fa.actor_id = a.actor_id
    ->     AND f.rating = 'R');
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JANE       | JACKMAN   |
+------------+-----------+
1 row in set (0.00 sec)
```

This query finds all actors who have never appeared in an R-rated film.

Data Manipulation Using Correlated Subqueries

All of the examples thus far in the chapter have been `select` statements, but don’t think that means that subqueries aren’t useful in other SQL statements. Subqueries are used heavily in `update`, `delete`, and `insert` statements as well, with correlated subqueries appearing frequently in `update` and `delete` statements. Here's an example of a correlated subquery used to modify the `last_update` column in the `customer` table:

```
UPDATE customer c
SET c.last_update =
 (SELECT max(r.rental_date) FROM rental r
  WHERE r.customer_id = c.customer_id);
```

This statement modifies every row in the `customer` table (since there is no `where` clause) by finding the latest rental date for each customer in the `rental` table. While it seems reasonable to expect that every customer will have at least one film rental, it would be best to check before attempting to update the `last_update` column; otherwise, the column will be set to `null`, since the subquery would return no rows. Here's another version of the `update` statement, this time employing a `where` clause with a second correlated subquery:

```
UPDATE customer c
SET c.last_update =
 (SELECT max(r.rental_date) FROM rental r
  WHERE r.customer_id = c.customer_id)
WHERE EXISTS
 (SELECT 1 FROM rental r
  WHERE r.customer_id = c.customer_id);
```

The two correlated subqueries are identical except for the `select` clauses. The subquery in the `set` clause, however, executes only if the condition in the `update` statement's `where` clause evaluates to `true` (meaning that at least one rental was found for the customer), thus protecting the data in the `last_update` column from being overwritten with a `null`.

Correlated subqueries are also common in `delete` statements. For example, you may run a data maintenance script at the end of each month that removes unnecessary data. The script might include the following statement, which removes rows from the `customer` table where there have been no film rentals in the past year:

```
DELETE FROM customer
WHERE 365 <
 (SELECT datediff(now(), r.rental_date)
days_since_last_rental
  FROM rental r
  WHERE r.customer_id = customer.customer_id);
```

When using correlated subqueries with `delete` statements in MySQL, keep in mind that, for whatever reason, table aliases are not allowed when using `delete`, which is why I had to use the entire table name in the subquery. With most other database servers, you could provide an alias for the `customer` table, such as:

```
DELETE FROM customer c
WHERE 365 <
 (SELECT datediff(now(), r.rental_date)
days_since_last_rental
  FROM rental r
  WHERE r.customer_id = c.customer_id);
```

When to Use Subqueries

Now that you have learned about the different types of subqueries and the different operators that you can employ to interact with the data returned by subqueries, it’s time to explore the many ways in which you can use subqueries to build powerful SQL statements. The next three sections demonstrate how you may use subqueries to construct custom tables, to build conditions, and to generate column values in result sets.

Subqueries As Data Sources

Back in Chapter 3, I stated that the `from` clause of a `select` statement contains the *tables* to be used by the query. Since a subquery generates a result set containing rows and columns of data, it is perfectly valid to include subqueries in your `from` clause along with tables. Although it might, at first glance, seem like an interesting feature without much practical merit, using subqueries alongside tables is one of the most powerful tools available when writing queries. Here’s a simple example:

```
mysql> SELECT c.first_name, c.last_name,
    ->   pymnt.num_rentals, pymnt.tot_payments
    -> FROM customer c
    ->   INNER JOIN
    ->    (SELECT customer_id,
    ->       count(*) num_rentals, sum(amount) tot_payments
    ->     FROM payment
    ->     GROUP BY customer_id
    ->    ) pymnt
    ->   ON c.customer_id = pymnt.customer_id;
+-------------+--------------+-------------+--------------+
| first_name  | last_name    | num_rentals | tot_payments |
+-------------+--------------+-------------+--------------+
| MARY        | SMITH        |          32 |       118.68 |
| PATRICIA    | JOHNSON      |          27 |       128.73 |
| LINDA       | WILLIAMS     |          26 |       135.74 |
| BARBARA     | JONES        |          22 |        81.78 |
| ELIZABETH   | BROWN        |          38 |       144.62 |
...
| TERRENCE    | GUNDERSON    |          30 |       117.70 |
```

```
| ENRIQUE     | FORSYTHE     |          28 |        96.72 |
| FREDDIE     | DUGGAN       |          25 |        99.75 |
| WADE        | DELVALLE     |          22 |        83.78 |
| AUSTIN      | CINTRON      |          19 |        83.81 |
+-------------+--------------+-------------+--------------+
599 rows in set (0.03 sec)
```

In this example, a subquery generates a list of customer IDs along with the number of film rentals and the total payments. Here's the result set generated by the subquery:

```
mysql> SELECT customer_id, count(*) num_rentals, sum(amount)
tot_payments
    -> FROM payment
    -> GROUP BY customer_id;
+-------------+-------------+--------------+
| customer_id | num_rentals | tot_payments |
+-------------+-------------+--------------+
|           1 |          32 |       118.68 |
|           2 |          27 |       128.73 |
|           3 |          26 |       135.74 |
|           4 |          22 |        81.78 |
...
|         596 |          28 |        96.72 |
|         597 |          25 |        99.75 |
|         598 |          22 |        83.78 |
|         599 |          19 |        83.81 |
+-------------+-------------+--------------+
599 rows in set (0.03 sec)
```

The subquery is given the name `pymnt` and is joined to the `customer` table via the `customer_id` column. The containing query then retrieves the customer's name from the `customer` table, along with the summary columns from the `pymnt` subquery.

Subqueries used in the `from` clause must be noncorrelated[1]; they are executed first, and the data is held in memory until the containing query finishes execution. Subqueries offer immense flexibility when writing queries, because you can go far beyond the set of available tables to create virtually any view of the data that you desire, and then join the results to other tables or subqueries. If you are writing reports or generating data feeds to external systems, you may be able to do things with a single query that used to demand multiple queries or a procedural language to accomplish.

DATA FABRICATION

Along with using subqueries to summarize existing data, you can use subqueries to generate data that doesn't exist in any form within your database. For example, you may wish to group your customers by the amount of money spent on film rentals, but you want to use group definitions that are not stored in your database. For example, let's say you want to sort your customers into the groups shown in Table 9-1.

Table 9-1. Customer payment groups

Group name	Lower limit	Upper limit
Small Fry	0	$74.99
Average Joes	$75	$149.99
Heavy Hitters	$150	$9,999,999.99

To generate these groups within a single query, you will need a way to define these three groups. The first step is to define a query that generates the group definitions:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 74.99
high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 75 low_limit, 149.99
high_limit
    -> UNION ALL
    -> SELECT 'Heavy Hitters' name, 150 low_limit,
9999999.99 high_limit;
+---------------+-----------+------------+
| name          | low_limit | high_limit |
+---------------+-----------+------------+
| Small Fry     |         0 |      74.99 |
| Average Joes  |        75 |     149.99 |
| Heavy Hitters |       150 | 9999999.99 |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

I have used the set operator `union all` to merge the results from three separate queries into a single result set. Each query retrieves three literals, and the results from the three queries are put together to generate a result set with three rows and three columns. You now have a query to generate the desired groups, and you can place it into the `from` clause of another query to generate your customer groups:

```
mysql> SELECT pymnt_grps.name, count(*) num_customers
    -> FROM
    ->  (SELECT customer_id,
    ->     count(*) num_rentals, sum(amount) tot_payments
    ->   FROM payment
    ->   GROUP BY customer_id
    ->  ) pymnt
    ->   INNER JOIN
    ->  (SELECT 'Small Fry' name, 0 low_limit, 74.99
high_limit
    ->   UNION ALL
    ->   SELECT 'Average Joes' name, 75 low_limit, 149.99
high_limit
    ->   UNION ALL
    ->   SELECT 'Heavy Hitters' name, 150 low_limit,
9999999.99 high_limit
    ->  ) pymnt_grps
    ->   ON pymnt.tot_payments
    ->     BETWEEN pymnt_grps.low_limit AND
pymnt_grps.high_limit
    -> GROUP BY pymnt_grps.name;
+---------------+---------------+
| name          | num_customers |
+---------------+---------------+
| Average Joes  |           515 |
| Heavy Hitters |            46 |
| Small Fry     |            38 |
+---------------+---------------+
3 rows in set (0.03 sec)
```

The `from` clause contains two subqueries; the first subquery, named `pymnt`, returns the total number of film rentals and total payments for each customer, while the second subquery, named `pymnt_grps`, generates the three customer groupings. The two subqueries are joined by finding which of the 3 groups each customer belongs to, and the rows are then grouped by the Group Name in order to count the number of customers in each group.

Of course, you could simply decide to build a permanent (or temporary) table to hold the group definitions instead of using a subquery. Using that approach, you would find your database to be littered with small special-purpose tables after awhile, and you wouldn't remember the reason for which most of them were created. Using subqueries, however, you will be able to adhere to a policy where tables are added to a database only when there is a clear business need to store new data.

TASK-ORIENTED SUBQUERIES

Let's say that you want to generate a report showing each customer's name, along with their city, the total number of rentals, and the total payment amount. You could accomplish this by joining the `payment`, `customer`, `address`, and `city` tables, and then grouping on the customer's first and last names:

```
mysql> SELECT c.first_name, c.last_name, ct.city,
    ->   sum(p.amount) tot_payments, count(*) tot_rentals
    -> FROM payment p
    ->   INNER JOIN customer c
    ->   ON p.customer_id = c.customer_id
    ->   INNER JOIN address a
    ->   ON c.address_id = a.address_id
    ->   INNER JOIN city ct
    ->   ON a.city_id = ct.city_id
    -> GROUP BY c.first_name, c.last_name, ct.city;
+-------------+-----------+-----------------+-------------
+-------------+
| first_name  | last_name  | city            | tot_payments
```

```
| tot_rentals |
+-------------+------------+----------------+-------------
+-------------+
| MARY        | SMITH      | Sasebo         |       118.68
|          32 |
| PATRICIA    | JOHNSON    | San Bernardino |       128.73
|          27 |
| LINDA       | WILLIAMS   | Athenai        |       135.74
|          26 |
| BARBARA     | JONES      | Myingyan       |        81.78
|          22 |
...
| TERRENCE    | GUNDERSON  | Jinzhou        |       117.70
|          30 |
| ENRIQUE     | FORSYTHE   | Patras         |        96.72
|          28 |
| FREDDIE     | DUGGAN     | Sullana        |        99.75
|          25 |
| WADE        | DELVALLE   | Lausanne       |        83.78
|          22 |
| AUSTIN      | CINTRON    | Tieli          |        83.81
|          19 |
+-------------+------------+----------------+-------------
+-------------+
599 rows in set (0.06 sec)
```

This query returns the desired data, but if you look at the query closely, you will see that the `customer`, `address`, and `city` tables are needed only for display purposes, and that the `payment` table has everything needed to generate the groupings (`customer_id` and `amount`). Therefore, you could separate out the task of generating the groups into a subquery, and then join the other three tables to the table generated by the subquery to achieve the desired end result. Here's the grouping subquery:

```
mysql> SELECT customer_id,
    ->   count(*) tot_rentals, sum(amount) tot_payments
    -> FROM payment
    -> GROUP BY customer_id;
+-------------+-------------+--------------+
| customer_id | tot_rentals | tot_payments |
```

```
+-------------+------------+-------------+
|           1 |         32 |      118.68 |
|           2 |         27 |      128.73 |
|           3 |         26 |      135.74 |
|           4 |         22 |       81.78 |
...
|         595 |         30 |      117.70 |
|         596 |         28 |       96.72 |
|         597 |         25 |       99.75 |
|         598 |         22 |       83.78 |
|         599 |         19 |       83.81 |
+-------------+------------+-------------+
599 rows in set (0.03 sec)
```

This is the heart of the query; the other tables are needed only to provide meaningful strings in place of the `customer_id` value. The next query joins the previous data set to the other three tables:

```
mysql> SELECT c.first_name, c.last_name,
    ->   ct.city,
    ->   pymnt.tot_payments, pymnt.tot_rentals
    -> FROM
    ->  (SELECT customer_id,
    ->     count(*) tot_rentals, sum(amount) tot_payments
    ->   FROM payment
    ->   GROUP BY customer_id
    ->  ) pymnt
    ->   INNER JOIN customer c
    ->   ON pymnt.customer_id = c.customer_id
    ->   INNER JOIN address a
    ->   ON c.address_id = a.address_id
    ->   INNER JOIN city ct
    ->   ON a.city_id = ct.city_id;
+-------------+------------+-----------------+--------------
+-------------+
| first_name  | last_name  | city            | tot_payments
| tot_rentals |
+-------------+------------+-----------------+--------------
+-------------+
| MARY        | SMITH      | Sasebo          |       118.68
|          32 |
| PATRICIA    | JOHNSON    | San Bernardino  |       128.73
|          27 |
| LINDA       | WILLIAMS   | Athenai         |       135.74
|          26 |
```

```
| BARBARA      | JONES      | Myingyan        |         81.78
|          22 |
...
| TERRENCE     | GUNDERSON  | Jinzhou         |        117.70
|          30 |
| ENRIQUE      | FORSYTHE   | Patras          |         96.72
|          28 |
| FREDDIE      | DUGGAN     | Sullana         |         99.75
|          25 |
| WADE         | DELVALLE   | Lausanne        |         83.78
|          22 |
| AUSTIN       | CINTRON    | Tieli           |         83.81
|          19 |
+-------------+------------+-----------------+-------------
+-------------+
599 rows in set (0.06 sec)
```

I realize that beauty is in the eye of the beholder, but I find this version of the query to be far more satisfying than the big, flat version. This version may execute faster as well, because the grouping is being done on a single numeric column(`customer_id`) instead of multiple lengthy string columns (`customer.first_name`, `customer.last_name`, `city.city`).

COMMON TABLE EXPRESSIONS

Common table expressions (a.k.a., CTEs), which are new to MySQL in version 8.0, have been available in other database servers for quite some time. A CTE is a named subquery which appears at the top of a query in a *with clause*, which can contain multiple CTEs separated by commas. Along with making queries more understandable, this feature also allows each subquery to refer to any other subquery defined previously. Here’s an example which includes three subqueries, where the second subquery refers to the 1st, and the 3rd refers to the 2nd:

```
mysql> WITH actors_s AS
    ->  (SELECT actor_id, first_name, last_name
    ->   FROM actor
    ->   WHERE last_name LIKE 'S%'
    ->  ),
    ->  actors_s_pg AS
    ->  (SELECT s.actor_id, s.first_name, s.last_name,
    ->     f.film_id, f.title
    ->   FROM actors_s s
    ->     INNER JOIN film_actor fa
    ->     ON s.actor_id = fa.actor_id
    ->     INNER JOIN film f
    ->     ON f.film_id = fa.film_id
    ->   WHERE f.rating = 'PG'
    ->  ),
    ->  actors_s_pg_revenue AS
    ->  (SELECT spg.first_name, spg.last_name, p.amount
    ->   FROM actors_s_pg spg
    ->     INNER JOIN inventory i
    ->     ON i.film_id = spg.film_id
    ->     INNER JOIN rental r
    ->     ON i.inventory_id = r.inventory_id
    ->     INNER JOIN payment p
    ->     ON r.rental_id = p.rental_id
    ->  )
    -> SELECT spg_rev.first_name, spg_rev.last_name,
    ->   sum(spg_rev.amount) tot_revenue
    -> FROM actors_s_pg_revenue spg_rev
    -> GROUP BY spg_rev.first_name, spg_rev.last_name
    -> ORDER BY 3 desc;
+------------+-------------+-------------+
| first_name | last_name   | tot_revenue |
+------------+-------------+-------------+
| NICK       | STALLONE    |      692.21 |
| JEFF       | SILVERSTONE |      652.35 |
| DAN        | STREEP      |      509.02 |
| GROUCHO    | SINATRA     |      457.97 |
| SISSY      | SOBIESKI    |      379.03 |
| JAYNE      | SILVERSTONE |      372.18 |
| CAMERON    | STREEP      |      361.00 |
| JOHN       | SUVARI      |      296.36 |
| JOE        | SWANK       |      177.52 |
+------------+-------------+-------------+
9 rows in set (0.18 sec)
```

This query calculates the total revenues generated from PG-rated film rentals where the cast included an actor whose last name starts with S. The first subquery (actors_s) finds all actors whose last name starts with S, the second subquery (actors_s_pg) joins that data set to the `film` table and filters on films having a PG rating, and the third subquery (actors_s_pg_revenue) joins that data set to the `revenue` table to generate the amounts paid to rent any of these films. The final query simply groups the data by the actors names and sums the revenues.

Subqueries As Expression Generators

For this last section of the chapter, I finish where I began: with single-column, single-row scalar subqueries. Along with being used in filter conditions, scalar subqueries may be used wherever an expression can appear, including the `select` and `order by` clauses of a query and the `values` clause of an `insert` statement.

In "Task-oriented subqueries", I showed you how to use a subquery to separate out the grouping mechanism from the rest of the query. Here's another version of the same query that uses subqueries for the same purpose, but in a different way:

```
mysql> SELECT
    ->  (SELECT c.first_name FROM customer c
    ->   WHERE c.customer_id = p.customer_id
    ->  ) first_name,
    ->  (SELECT c.last_name FROM customer c
    ->   WHERE c.customer_id = p.customer_id
    ->  ) last_name,
    ->  (SELECT ct.city
    ->   FROM customer c
    ->   INNER JOIN address a
```

```
    ->       ON c.address_id = a.address_id
    ->     INNER JOIN city ct
    ->       ON a.city_id = ct.city_id
    ->     WHERE c.customer_id = p.customer_id
    ->   ) city,
    ->   sum(p.amount) tot_payments,
    ->   count(*) tot_rentals
    -> FROM payment p
    -> GROUP BY p.customer_id;
+-------------+------------+-----------------+-------------
+-------------+
| first_name  | last_name  | city            | tot_payments
| tot_rentals |
+-------------+------------+-----------------+-------------
+-------------+
| MARY        | SMITH      | Sasebo          |       118.68
|          32 |
| PATRICIA    | JOHNSON    | San Bernardino  |       128.73
|          27 |
| LINDA       | WILLIAMS   | Athenai         |       135.74
|          26 |
| BARBARA     | JONES      | Myingyan        |        81.78
|          22 |
...
| TERRENCE    | GUNDERSON  | Jinzhou         |       117.70
|          30 |
| ENRIQUE     | FORSYTHE   | Patras          |        96.72
|          28 |
| FREDDIE     | DUGGAN     | Sullana         |        99.75
|          25 |
| WADE        | DELVALLE   | Lausanne        |        83.78
|          22 |
| AUSTIN      | CINTRON    | Tieli           |        83.81
|          19 |
+-------------+------------+-----------------+-------------
+-------------+
599 rows in set (0.06 sec)
```

There are two main differences between this query and the earlier version using a subquery in the `from` clause:

- Instead of joining the `customer`, `address`, and `city` tables to the payment data, correlated scalar subqueries are used in the `select` clause to look up the customer's first/last names and city.

- The `customer` table is accessed three times (once in each of the three subqueries) rather than just once.

The `customer` table is accessed three times because scalar subqueries can only return a single column and row, so if we need three columns related to the customer, it is necessary to use three different subqueries.

As previously noted, scalar subqueries can also appear in the `order by` clause. The following query retrieves actor's first and last names and sorts by the number of films in which the actor appeared:

```
mysql> SELECT a.actor_id, a.first_name, a.last_name
    -> FROM actor a
    -> ORDER BY
    ->  (SELECT count(*) FROM film_actor fa
    ->   WHERE fa.actor_id = a.actor_id) DESC;
+----------+------------+-------------+
| actor_id | first_name | last_name   |
+----------+------------+-------------+
|      107 | GINA       | DEGENERES   |
|      102 | WALTER     | TORN        |
|      198 | MARY       | KEITEL      |
|      181 | MATTHEW    | CARREY      |
...
|       71 | ADAM       | GRANT       |
|      186 | JULIA      | ZELLWEGER   |
|       35 | JUDY       | DEAN        |
|      199 | JULIA      | FAWCETT     |
|      148 | EMILY      | DEE         |
+----------+------------+-------------+
200 rows in set (0.01 sec)
```

The query uses a correlated scalar subquery in the `order by` clause to return just the number of film appearances, and this value is used solely for sorting purposes.

Along with using correlated scalar subqueries in `select` statements, you can use noncorrelated scalar subqueries to generate values for an `insert` statement. For example, let's say you are going to generate a new row in the `film_actor` table, and you've been given the following data:

- The first and last name of the actor
- The name of the film

You have two choices for how to go about it: execute two queries to retrieve the primary key values from `film` and `actor` and place those values into an `insert` statement, or use subqueries to retrieve the two key values from within an `insert` statement. Here's an example of the latter approach:

```
INSERT INTO film_actor (actor_id, film_id, last_update)
VALUES (
 (SELECT actor_id FROM actor
  WHERE first_name = 'JENNIFER' AND last_name = 'DAVIS'),
 (SELECT film_id FROM film
  WHERE title = 'ACE GOLDFINGER'),
 now()
 );
```

Using a single SQL statement, you can create a row in the `film_actor` table and look up two foreign key column values at the same time.

Subquery Wrap-up

I covered a lot of ground in this chapter, so it might be a good idea to review it. The examples I used in this chapter demonstrated subqueries that:

- Return a single column and row, a single column with multiple rows, and multiple columns and rows
- Are independent of the containing statement (noncorrelated subqueries)
- Reference one or more columns from the containing statement (correlated subqueries)
- Are used in conditions that utilize comparison operators as well as the special-purpose operators `in`, `not in`, `exists`, and `not exists`
- Can be found in `select`, `update`, `delete`, and `insert` statements
- Generate result sets that can be joined to other tables (or subqueries) in a query
- Can be used to generate values to populate a table or to populate columns in a query’s result set
- Are used in the `select`, `from`, `where`, `having`, and `order by` clauses of queries

Obviously, subqueries are a very versatile tool, so don’t feel bad if all these concepts haven’t sunk in after reading this chapter for the first time. Keep experimenting with the various uses for subqueries, and you will soon find yourself thinking about how you might utilize a subquery every time you write a nontrivial SQL statement.

Test Your Knowledge

These exercises are designed to test your understanding of subqueries. Please see Appendix C for the solutions.

Exercise 9-1

Construct a query against the `film` table that uses a filter condition with a noncorrelated subquery against the `category` table to find all action films (`category.name = 'Action'`).

Exercise 9-2

Rework the query from Exercise 9-1 using a *correlated* subquery against the `category` and `film_category` tables to achieve the same results.

Exercise 9-3

Join the following query to a subquery against the `film_actor` table to show the level of each actor:

```
SELECT 'Hollywood Star' level, 30 min_roles, 99999 max_roles
UNION ALL
SELECT 'Prolific Actor' level, 20 min_roles, 29 max_roles
UNION ALL
SELECT 'Newcomer' level, 1 min_roles, 19 max_roles
```

The subquery against the `film_actor` table should count the number of rows for each actor using `group by actor_id`, and the count should be compared to the min_roles/max_roles columns to determine which level each actor belongs to.

1 Actually, depending on which database server you are using, you might be able to include correlated subqueries in your from clause by using Cross Apply or Outer Apply, but these features are beyond the scope of this book.

Chapter 10. Joins Revisited

By now, you should be comfortable with the concept of the inner join, which I introduced in Chapter 5. This chapter focuses on other ways in which you can join tables, including the outer join and the cross join.

Outer Joins

In all the examples thus far that have included multiple tables, we haven't been concerned that the join conditions might fail to find matches for all the rows in the tables. For example, the `inventory` table contains a row for every film available for rental, but of the 1,000 rows in the `film` table, only 958 have one or more rows in the `inventory` table. The other 42 films are not available for rental (perhaps they are new releases due to arrive in a few days), so these film IDs cannot be found in the `inventory` table. Here's a query that counts the number of available copies of each film by joining these two tables:

```
mysql> SELECT f.film_id, f.title, count(*) num_copies
    -> FROM film f
    ->   INNER JOIN inventory i
    ->   ON f.film_id = i.film_id
    -> GROUP BY f.film_id, f.title;
+---------+-----------------------------+------------+
| film_id | title                       | num_copies |
+---------+-----------------------------+------------+
|       1 | ACADEMY DINOSAUR            |          8 |
|       2 | ACE GOLDFINGER              |          3 |
```

```
|       3 | ADAPTATION HOLES             |          4 |
|       4 | AFFAIR PREJUDICE             |          7 |
...
|      13 | ALI FOREVER                  |          4 |
|      15 | ALIEN CENTER                 |          6 |
...
|     997 | YOUTH KICK                   |          2 |
|     998 | ZHIVAGO CORE                 |          2 |
|     999 | ZOOLANDER FICTION            |          5 |
|    1000 | ZORRO ARK                    |          8 |
+---------+------------------------------+------------+
958 rows in set (0.02 sec)
```

While you may have expected 1,000 rows to be returned (one for each film), the query only returns 958 rows. This is because the query uses an inner join, which only returns rows which satisfy the join condition. The film “ALICE FANTASIA” (film_id 14) doesn’t appear in the results, for example, because it doesn’t have any rows in the `inventory` table.

If you want the query to return all 1,000 films, regardless of whether or not there are rows in the `inventory` table, you can use an **outer** join, which essentially makes the join condition optional :

```
mysql> SELECT f.film_id, f.title, count(i.inventory_id)
num_copies
    -> FROM film f
    ->   LEFT OUTER JOIN inventory i
    ->   ON f.film_id = i.film_id
    -> GROUP BY f.film_id, f.title;
+---------+------------------------------+------------+
| film_id | title                        | num_copies |
+---------+------------------------------+------------+
|       1 | ACADEMY DINOSAUR             |          8 |
|       2 | ACE GOLDFINGER               |          3 |
|       3 | ADAPTATION HOLES             |          4 |
|       4 | AFFAIR PREJUDICE             |          7 |
...
|      13 | ALI FOREVER                  |          4 |
|      14 | ALICE FANTASIA               |          0 |
```

```
|      15 | ALIEN CENTER                  |          6 |
...
|     997 | YOUTH KICK                    |          2 |
|     998 | ZHIVAGO CORE                  |          2 |
|     999 | ZOOLANDER FICTION             |          5 |
|    1000 | ZORRO ARK                     |          8 |
+---------+-----------------------------+------------+
1000 rows in set (0.01 sec)
```

As you can see, the query now returns all 1,000 rows from the `film` table, and 42 of the rows (including “ALICE FANTASIA”) have a value of 0 in the `num_copies` column, which indicates that there are no copies in inventory.

Here’s a description of the changes from the prior version of the query:

- The join definition was changed from `inner` to `left outer`, which instructs the server to include all rows from the table on the left side of the join (`film`, in this case), and then include columns from the table on the right side of the join (`inventory`) if the join is successful
- The `num_copies` column definition was changed from count(*) to count(i.inventory_id), which will count the number of non-NULL values of the `inventory.inventory_id` column

Next, let’s remove the `group by` clause and filter out most of the rows in order to clearly see the differences between `inner` and `outer` joins. Here’s a query using an `inner` join and a filter condition to return rows for just a few films:

```
mysql> SELECT f.film_id, f.title, i.inventory_id
    -> FROM film f
```

```
    ->   INNER JOIN inventory i
    ->   ON f.film_id = i.film_id
    -> WHERE f.film_id BETWEEN 13 AND 15;
+---------+--------------+--------------+
| film_id | title        | inventory_id |
+---------+--------------+--------------+
|      13 | ALI FOREVER  |           67 |
|      13 | ALI FOREVER  |           68 |
|      13 | ALI FOREVER  |           69 |
|      13 | ALI FOREVER  |           70 |
|      15 | ALIEN CENTER |           71 |
|      15 | ALIEN CENTER |           72 |
|      15 | ALIEN CENTER |           73 |
|      15 | ALIEN CENTER |           74 |
|      15 | ALIEN CENTER |           75 |
|      15 | ALIEN CENTER |           76 |
+---------+--------------+--------------+
10 rows in set (0.00 sec)
```

The results show that there are 4 copies of "ALI FOREVER" and 6 copies of "ALIEN CENTER" in inventory. Here's the same query, but using an `outer` join:

```
mysql> SELECT f.film_id, f.title, i.inventory_id
    -> FROM film f
    ->   LEFT OUTER JOIN inventory i
    ->   ON f.film_id = i.film_id
    -> WHERE f.film_id BETWEEN 13 AND 15;
+---------+----------------+--------------+
| film_id | title          | inventory_id |
+---------+----------------+--------------+
|      13 | ALI FOREVER    |           67 |
|      13 | ALI FOREVER    |           68 |
|      13 | ALI FOREVER    |           69 |
|      13 | ALI FOREVER    |           70 |
|      14 | ALICE FANTASIA |         NULL |
|      15 | ALIEN CENTER   |           71 |
|      15 | ALIEN CENTER   |           72 |
|      15 | ALIEN CENTER   |           73 |
|      15 | ALIEN CENTER   |           74 |
|      15 | ALIEN CENTER   |           75 |
|      15 | ALIEN CENTER   |           76 |
+---------+----------------+--------------+
11 rows in set (0.00 sec)
```

The results are the same for “ALI FOREVER” and “ALIEN CENTER”, but there’s one new row for “ALICE FANTASIA”, with a NULL value for the `inventory.inventory_id` column. This example illustrates how an `outer` join will add column values without restricting the number of rows returned by the query.

Left Versus Right Outer Joins

In each of the outer join examples in the previous section, I specified `left outer join`. The keyword `left` indicates that the table on the left side of the join is responsible for determining the number of rows in the result set, whereas the table on the right side is used to provide column values whenever a match is found. However, you may also specify a `right outer join`, in which case the table on the right side of the join is responsible for determining the number of rows in the result set, whereas the table on the left side is used to provide column values.

Here’s the last query from the previous section, but rearranged to use a `right outer join` instead of a `left outer join`:

```
mysql> SELECT f.film_id, f.title, i.inventory_id
    -> FROM inventory i
    ->   RIGHT OUTER JOIN film f
    ->   ON f.film_id = i.film_id
    -> WHERE f.film_id BETWEEN 13 AND 15;
+---------+----------------+--------------+
| film_id | title          | inventory_id |
+---------+----------------+--------------+
|      13 | ALI FOREVER    |           67 |
|      13 | ALI FOREVER    |           68 |
|      13 | ALI FOREVER    |           69 |
|      13 | ALI FOREVER    |           70 |
|      14 | ALICE FANTASIA |         NULL |
|      15 | ALIEN CENTER   |           71 |
```

```
|      15 | ALIEN CENTER   |           72 |
|      15 | ALIEN CENTER   |           73 |
|      15 | ALIEN CENTER   |           74 |
|      15 | ALIEN CENTER   |           75 |
|      15 | ALIEN CENTER   |           76 |
+---------+----------------+--------------+
11 rows in set (0.00 sec)
```

Keep in mind that both versions of the query are performing outer joins; the keywords `left` and `right` are there just to tell the server which table is allowed to have gaps in the data. If you want to outer-join tables A and B and you want all rows from A with additional columns from B whenever there is matching data, you can specify either `A left outer join B` or `B right outer join A`.

Three-Way Outer Joins

In some cases, you may want to outer-join one table with two other tables. For example, the query from a prior section can be expanded to include data from the `rental` table:

```
mysql> SELECT f.film_id, f.title, i.inventory_id,
r.rental_date
    -> FROM film f
    ->   LEFT OUTER JOIN inventory i
    ->   ON f.film_id = i.film_id
    ->   LEFT OUTER JOIN rental r
    ->   ON i.inventory_id = r.inventory_id
    -> WHERE f.film_id BETWEEN 13 AND 15;
+---------+----------------+--------------+-----------------
----+
| film_id | title          | inventory_id |
rental_date         |
+---------+----------------+--------------+-----------------
----+
|      13 | ALI FOREVER    |           67 | 2005-07-31
18:11:17 |
|      13 | ALI FOREVER    |           67 | 2005-08-22
21:59:29 |
```

```
|       13 | ALI FOREVER    |            68 | 2005-07-28
15:26:20 |
|       13 | ALI FOREVER    |            68 | 2005-08-23
05:02:31 |
|       13 | ALI FOREVER    |            69 | 2005-08-01
23:36:10 |
|       13 | ALI FOREVER    |            69 | 2005-08-22
02:12:44 |
|       13 | ALI FOREVER    |            70 | 2005-07-12
10:51:09 |
|       13 | ALI FOREVER    |            70 | 2005-07-29
01:29:51 |
|       13 | ALI FOREVER    |            70 | 2006-02-14
15:16:03 |
|       14 | ALICE FANTASIA |          NULL |
NULL                |
|       15 | ALIEN CENTER   |            71 | 2005-05-28
02:06:37 |
|       15 | ALIEN CENTER   |            71 | 2005-06-17
16:40:03 |
|       15 | ALIEN CENTER   |            71 | 2005-07-11
05:47:08 |
|       15 | ALIEN CENTER   |            71 | 2005-08-02
13:58:55 |
|       15 | ALIEN CENTER   |            71 | 2005-08-23
05:13:09 |
|       15 | ALIEN CENTER   |            72 | 2005-05-27
22:49:27 |
|       15 | ALIEN CENTER   |            72 | 2005-06-19
13:29:28 |
|       15 | ALIEN CENTER   |            72 | 2005-07-07
23:05:53 |
|       15 | ALIEN CENTER   |            72 | 2005-08-01
05:55:13 |
|       15 | ALIEN CENTER   |            72 | 2005-08-20
15:11:48 |
|       15 | ALIEN CENTER   |            73 | 2005-07-06
15:51:58 |
|       15 | ALIEN CENTER   |            73 | 2005-07-30
14:48:24 |
|       15 | ALIEN CENTER   |            73 | 2005-08-20
22:32:11 |
|       15 | ALIEN CENTER   |            74 | 2005-07-27
00:15:18 |
|       15 | ALIEN CENTER   |            74 | 2005-08-23
19:21:22 |
|       15 | ALIEN CENTER   |            75 | 2005-07-09
02:58:41 |
```

```
|       15 | ALIEN CENTER   |            75 | 2005-07-29
23:52:01 |
|       15 | ALIEN CENTER   |            75 | 2005-08-18
21:55:01 |
|       15 | ALIEN CENTER   |            76 | 2005-06-15
08:01:29 |
|       15 | ALIEN CENTER   |            76 | 2005-07-07
18:31:50 |
|       15 | ALIEN CENTER   |            76 | 2005-08-01
01:49:36 |
|       15 | ALIEN CENTER   |            76 | 2005-08-17
07:26:47 |
+---------+---------------+-------------+----------------
----+
32 rows in set (0.01 sec)
```

The results include all rentals of all films in inventory, but the film "ALICE FANTASIA" has NULL values for the columns from both outer-joined tables.

Cross Joins

Back in Chapter 5, I introduced the concept of a Cartesian product, which is essentially the result of joining multiple tables without specifying any join conditions. Cartesian products are used fairly frequently by accident (e.g., forgetting to add the join condition to the `from` clause) but are not so common otherwise. If, however, you *do* intend to generate the Cartesian product of two tables, you should specify a *cross join*, as in:

```
mysql> SELECT c.name category_name, l.name language_name
    -> FROM category c
    ->   CROSS JOIN language l;
+---------------+---------------+
| category_name | language_name |
+---------------+---------------+
| Action        | English       |
| Action        | Italian       |
```

```
| Action        | Japanese      |
| Action        | Mandarin      |
| Action        | French        |
| Action        | German        |
| Animation     | English       |
| Animation     | Italian       |
| Animation     | Japanese      |
| Animation     | Mandarin      |
| Animation     | French        |
| Animation     | German        |
...
| Sports        | English       |
| Sports        | Italian       |
| Sports        | Japanese      |
| Sports        | Mandarin      |
| Sports        | French        |
| Sports        | German        |
| Travel        | English       |
| Travel        | Italian       |
| Travel        | Japanese      |
| Travel        | Mandarin      |
| Travel        | French        |
| Travel        | German        |
+---------------+---------------+
96 rows in set (0.00 sec)
```

This query generates the Cartesian product of the `category` and `language` tables, resulting in 96 rows (16 `category` rows × 6 `language` rows). But now that you know what a cross join is and how to specify it, what is it used for? Most SQL books will describe what a cross join is and then tell you that it is seldom useful, but I would like to share with you a situation in which I find the cross join to be quite helpful.

In Chapter 9, I discussed how to use subqueries to fabricate tables. The example I used showed how to build a three-row table that could be joined to other tables. Here's the fabricated table from the example:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 74.99
high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 75 low_limit, 149.99
high_limit
    -> UNION ALL
    -> SELECT 'Heavy Hitters' name, 150 low_limit,
9999999.99 high_limit;
+---------------+-----------+------------+
| name          | low_limit | high_limit |
+---------------+-----------+------------+
| Small Fry     |         0 |      74.99 |
| Average Joes  |        75 |     149.99 |
| Heavy Hitters |       150 | 9999999.99 |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

While this table was exactly what was needed for placing customers into three groups based on their total film payments, this strategy of merging single-row tables using the set operator `union all` doesn't work very well if you need to fabricate a large table.

Say, for example, that you want to create a query that generates a row for every day in the year 2020, but you don't have a table in your database that contains a row for every day. Using the strategy from the example in Chapter 9, you could do something like the following:

```
SELECT '2020-01-01' dt
UNION ALL
SELECT '2020-01-02' dt
UNION ALL
SELECT '2020-01-03' dt
UNION ALL
...
...
...
SELECT '2020-12-29' dt
UNION ALL
SELECT '2020-12-30' dt
UNION ALL
SELECT '2020-12-31' dt
```

Building a query that merges together the results of 366 queries is a bit tedious, so maybe a different strategy is needed. What if you generate a table with 366 rows (2020 is a leap year) with a single column containing a number between 0 and 366, and then add that number of days to January 1, 2020? Here's one possible method to generate such a table:

```
mysql> SELECT ones.num + tens.num + hundreds.num
    -> FROM
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 1 num UNION ALL
    ->   SELECT 2 num UNION ALL
    ->   SELECT 3 num UNION ALL
    ->   SELECT 4 num UNION ALL
    ->   SELECT 5 num UNION ALL
    ->   SELECT 6 num UNION ALL
    ->   SELECT 7 num UNION ALL
    ->   SELECT 8 num UNION ALL
    ->   SELECT 9 num) ones
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 10 num UNION ALL
    ->   SELECT 20 num UNION ALL
    ->   SELECT 30 num UNION ALL
    ->   SELECT 40 num UNION ALL
    ->   SELECT 50 num UNION ALL
    ->   SELECT 60 num UNION ALL
    ->   SELECT 70 num UNION ALL
    ->   SELECT 80 num UNION ALL
    ->   SELECT 90 num) tens
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 100 num UNION ALL
    ->   SELECT 200 num UNION ALL
    ->   SELECT 300 num) hundreds;
+------------------------------------+
| ones.num + tens.num + hundreds.num |
+------------------------------------+
|                                  0 |
|                                  1 |
|                                  2 |
|                                  3 |
|                                  4 |
```

```
|                               5 |
|                               6 |
|                               7 |
|                               8 |
|                               9 |
|                              10 |
|                              11 |
|                              12 |
...
...
...
|                             391 |
|                             392 |
|                             393 |
|                             394 |
|                             395 |
|                             396 |
|                             397 |
|                             398 |
|                             399 |
+---------------------------------+
400 rows in set (0.00 sec)
```

If you take the Cartesian product of the three sets {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {0, 10, 20, 30, 40, 50, 60, 70, 80, 90}, and {0, 100, 200, 300} and add the values in the three columns, you get a 400-row result set containing all numbers between 0 and 399. While this is more than the 366 rows needed to generate the set of days in 2020, it's easy enough to get rid of the excess rows, and I'll show you how shortly.

The next step is to convert the set of numbers to a set of dates. To do this, I will use the `date_add()` function to add each number in the result set to January 1, 2020. Then I'll add a filter condition to throw away any dates that venture into 2021:

```
mysql> SELECT DATE_ADD('2020-01-01',
    ->   INTERVAL (ones.num + tens.num + hundreds.num) DAY)
dt
```

```
    -> FROM
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 1 num UNION ALL
    ->   SELECT 2 num UNION ALL
    ->   SELECT 3 num UNION ALL
    ->   SELECT 4 num UNION ALL
    ->   SELECT 5 num UNION ALL
    ->   SELECT 6 num UNION ALL
    ->   SELECT 7 num UNION ALL
    ->   SELECT 8 num UNION ALL
    ->   SELECT 9 num) ones
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 10 num UNION ALL
    ->   SELECT 20 num UNION ALL
    ->   SELECT 30 num UNION ALL
    ->   SELECT 40 num UNION ALL
    ->   SELECT 50 num UNION ALL
    ->   SELECT 60 num UNION ALL
    ->   SELECT 70 num UNION ALL
    ->   SELECT 80 num UNION ALL
    ->   SELECT 90 num) tens
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 100 num UNION ALL
    ->   SELECT 200 num UNION ALL
    ->   SELECT 300 num) hundreds
    -> WHERE DATE_ADD('2020-01-01',
    ->   INTERVAL (ones.num + tens.num + hundreds.num) DAY)
< '2021-01-01'
    -> ORDER BY 1;
+------------+
| dt         |
+------------+
| 2020-01-01 |
| 2020-01-02 |
| 2020-01-03 |
| 2020-01-04 |
| 2020-01-05 |
| 2020-01-06 |
| 2020-01-07 |
| 2020-01-08 |
...
...
...
| 2020-02-26 |
| 2020-02-27 |
| 2020-02-28 |
```

```
| 2020-02-29 |
| 2020-03-01 |
| 2020-03-02 |
| 2020-03-03 |
...
...
...
| 2020-12-24 |
| 2020-12-25 |
| 2020-12-26 |
| 2020-12-27 |
| 2020-12-28 |
| 2020-12-29 |
| 2020-12-30 |
| 2020-12-31 |
+------------+
366 rows in set (0.03 sec)
```

The nice thing about this approach is that the result set automatically includes the extra leap day (February 29) without your intervention, since the database server figures it out when it adds 59 days to January 1, 2020.

Now that you have a mechanism for fabricating all the days in 2020, what should you do with it? Well, you might be asked to generate a report that shows every day in 2020 along with the number of film rentals on that day. The report needs to include every day of the year, including days when no films are rented. Here's what the query might look like, but using the year 2005 to match the data in the `rental` table:

```
mysql> SELECT days.dt, COUNT(r.rental_id) num_rentals
    -> FROM rental r
    ->   RIGHT OUTER JOIN
    ->  (SELECT DATE_ADD('2005-01-01',
    ->     INTERVAL (ones.num + tens.num + hundreds.num)
DAY) dt
    ->   FROM
    ->    (SELECT 0 num UNION ALL
```

```
    ->     SELECT 1 num UNION ALL
    ->     SELECT 2 num UNION ALL
    ->     SELECT 3 num UNION ALL
    ->     SELECT 4 num UNION ALL
    ->     SELECT 5 num UNION ALL
    ->     SELECT 6 num UNION ALL
    ->     SELECT 7 num UNION ALL
    ->     SELECT 8 num UNION ALL
    ->     SELECT 9 num) ones
    ->     CROSS JOIN
    ->    (SELECT 0 num UNION ALL
    ->     SELECT 10 num UNION ALL
    ->     SELECT 20 num UNION ALL
    ->     SELECT 30 num UNION ALL
    ->     SELECT 40 num UNION ALL
    ->     SELECT 50 num UNION ALL
    ->     SELECT 60 num UNION ALL
    ->     SELECT 70 num UNION ALL
    ->     SELECT 80 num UNION ALL
    ->     SELECT 90 num) tens
    ->     CROSS JOIN
    ->    (SELECT 0 num UNION ALL
    ->     SELECT 100 num UNION ALL
    ->     SELECT 200 num UNION ALL
    ->     SELECT 300 num) hundreds
    ->   WHERE DATE_ADD('2005-01-01',
    ->     INTERVAL (ones.num + tens.num + hundreds.num)
DAY)
    ->       < '2006-01-01'
    ->  ) days
    ->   ON days.dt = date(r.rental_date)
    -> GROUP BY days.dt
    -> ORDER BY 1;
+------------+------------+
| dt         | num_rentals |
+------------+------------+
| 2005-01-01 |          0 |
| 2005-01-02 |          0 |
| 2005-01-03 |          0 |
| 2005-01-04 |          0 |
...
| 2005-05-23 |          0 |
| 2005-05-24 |          8 |
| 2005-05-25 |        137 |
| 2005-05-26 |        174 |
| 2005-05-27 |        166 |
| 2005-05-28 |        196 |
| 2005-05-29 |        154 |
```

```
| 2005-05-30 |         158 |
| 2005-05-31 |         163 |
| 2005-06-01 |           0 |
...
| 2005-06-13 |           0 |
| 2005-06-14 |          16 |
| 2005-06-15 |         348 |
| 2005-06-16 |         324 |
| 2005-06-17 |         325 |
| 2005-06-18 |         344 |
| 2005-06-19 |         348 |
| 2005-06-20 |         331 |
| 2005-06-21 |         275 |
| 2005-06-22 |           0 |
...
| 2005-12-27 |           0 |
| 2005-12-28 |           0 |
| 2005-12-29 |           0 |
| 2005-12-30 |           0 |
| 2005-12-31 |           0 |
+------------+-------------+
365 rows in set (8.99 sec)
```

This is one of the more interesting queries thus far in the book, in that it includes cross joins, outer joins, a date function, grouping, set operations (`union all`), and an aggregate function (`count()`). It is also not the most elegant solution to the given problem, but it should serve as an example of how, with a little creativity and a firm grasp on the language, you can make even a seldom-used feature like cross joins a potent tool in your SQL toolkit.

Natural Joins

If you are lazy (and aren't we all), you can choose a join type that allows you to name the tables to be joined but lets the database server determine what the join conditions need to be. Known as the *natural join*, this join type relies on identical column names across multiple tables to infer the proper join conditions. For example, the `rental` table includes a column named `customer_id`, which is the foreign key to the `customer` table, whose primary key is also named `customer_id`. Thus, you could try to write a query that uses `natural join` to join the two tables:

```
mysql> SELECT c.first_name, c.last_name, date(r.rental_date)
    -> FROM customer c
    ->   NATURAL JOIN rental r;
Empty set (0.04 sec)
```

Because you specified a natural join, the server inspected the table definitions and added the join condition `r.customer_id = c.customer_id` to join the two tables. This would have worked fine, but in the Sakila schema all of the tables include the column `last_update` to show when each row was last modified, so the server is also adding the join condition `r.last_update = c.last_update` , which causes the query to return no data.

The only way around this issue is to use a subquery to restrict the columns for at least one of the tables:

```
mysql> SELECT cust.first_name, cust.last_name,
date(r.rental_date)
    -> FROM
    ->  (SELECT customer_id, first_name, last_name
    ->   FROM customer
    ->  ) cust
    ->   NATURAL JOIN rental r;
+------------+-----------+---------------------+
| first_name | last_name | date(r.rental_date) |
+------------+-----------+---------------------+
| MARY       | SMITH     | 2005-05-25          |
| MARY       | SMITH     | 2005-05-28          |
| MARY       | SMITH     | 2005-06-15          |
| MARY       | SMITH     | 2005-06-15          |
```

```
| MARY        | SMITH      | 2005-06-15          |
| MARY        | SMITH      | 2005-06-16          |
| MARY        | SMITH      | 2005-06-18          |
| MARY        | SMITH      | 2005-06-18          |
...
| AUSTIN      | CINTRON    | 2005-08-21          |
| AUSTIN      | CINTRON    | 2005-08-21          |
| AUSTIN      | CINTRON    | 2005-08-21          |
| AUSTIN      | CINTRON    | 2005-08-23          |
| AUSTIN      | CINTRON    | 2005-08-23          |
| AUSTIN      | CINTRON    | 2005-08-23          |
+-------------+------------+---------------------+
16044 rows in set (0.03 sec)
```

So, is the reduced wear and tear on the old fingers from not having to type the join condition worth the trouble? Absolutely not; you should avoid this join type and use inner joins with explicit join conditions.

Test Your Knowledge

The following exercises test your understanding of outer and cross joins. Please see Appendix C for solutions.

Exercise 10-1

Using the table definitions and data below, write a query that returns each customer name along with their total balance across all accounts.

```
                        Customer:
Customer_id     Name
-----------     ---------------
1               John Smith
2               Kathy Jones
3               Greg Oliver

                        Account:
Account_id      Customer_id     Account_Name    Balance
----------      -----------     ------------    --------
```

```
101            1              Checking        1044
102            3              Savings         522
103            1              Line of Credit  9995
```

Include all customers, even if no accounts exist for that customer.

Exercise 10-2

Reformulate your query from Exercise 10-1 to use the other outer join type (e.g., if you used a left outer join in Exercise 10-1, use a right outer join this time) such that the results are identical to Exercise 10-1.

Exercise 10-3 (Extra Credit)

Devise a query that will generate the set {1, 2, 3,…, 99, 100}. (Hint: use a cross join with at least two `from` clause subqueries.)

Chapter 11. Conditional Logic

In certain situations, you may want your SQL logic to branch in one direction or another depending on the values of certain columns or expressions. This chapter focuses on how to write statements that can behave differently depending on the data encountered during statement execution.

What Is Conditional Logic?

Conditional logic is simply the ability to take one of several paths during program execution. For example, when querying customer information, you might want to include the `customer.active` column, which stores 1 to indicate Active and 0 to indicate Inactive. If the query results are being used to generate a report, you may want to translate the value to improve readability. While every database includes built-in functions for these types of situations, there are no standards, so you would need to remember which functions are used by which database. Fortunately, every database's SQL implementation includes the CASE expression, which is useful in many situations, including simple translations :

```
mysql> SELECT first_name, last_name,
    ->   CASE
    ->     WHEN active = 1 THEN 'ACTIVE'
    ->     ELSE 'INACTIVE'
```

```
    ->   END activity_type
    -> FROM customer;
+-------------+-------------+---------------+
| first_name  | last_name   | activity_type |
+-------------+-------------+---------------+
| MARY        | SMITH       | ACTIVE        |
| PATRICIA    | JOHNSON     | ACTIVE        |
| LINDA       | WILLIAMS    | ACTIVE        |
| BARBARA     | JONES       | ACTIVE        |
| ELIZABETH   | BROWN       | ACTIVE        |
| JENNIFER    | DAVIS       | ACTIVE        |
...
| KENT        | ARSENAULT   | ACTIVE        |
| TERRANCE    | ROUSH       | INACTIVE      |
| RENE        | MCALISTER   | ACTIVE        |
| EDUARDO     | HIATT       | ACTIVE        |
| TERRENCE    | GUNDERSON   | ACTIVE        |
| ENRIQUE     | FORSYTHE    | ACTIVE        |
| FREDDIE     | DUGGAN      | ACTIVE        |
| WADE        | DELVALLE    | ACTIVE        |
| AUSTIN      | CINTRON     | ACTIVE        |
+-------------+-------------+---------------+
599 rows in set (0.00 sec)
```

This query includes a CASE expression to generate a value for the `activity_type` column, which returns the strings “ACTIVE” or “INACTIVE” depending on the value of the `customer.active` column.

The Case Expression

All of the major database servers include built-in functions designed to mimic the if-then-else statement found in most programming languages (examples include Oracle’s `decode()` function, MySQL’s `if()` function, and SQL Server’s `coalesce()` function). Case expressions are also designed to facilitate if-then-else logic but enjoy two advantages over built-in functions:

- The case expression is part of the SQL standard (SQL92 release) and has been implemented by Oracle Database, SQL Server, MySQL, Sybase, PostgreSQL, IBM UDB, and others.
- Case expressions are built into the SQL grammar and can be included in `select`, `insert`, `update`, and `delete` statements.

The next two subsections introduce the two different types of case expressions, and then I show you some examples of case expressions in action.

Searched Case Expressions

The case expression demonstrated earlier in the chapter is an example of a *searched case expression*, which has the following syntax:

```
CASE
  WHEN C1 THEN E1
  WHEN C2 THEN E2
  ...
  WHEN CN THEN EN
  [ELSE ED]
END
```

In the previous definition, the symbols `C1`, `C2`,…, `CN` represent conditions, and the symbols `E1`, `E2`,…, `EN` represent expressions to be returned by the case expression. If the condition in a `when` clause evaluates to `true`, then the case expression returns the corresponding expression. Additionally, the `ED` symbol represents the default expression, which the case expression returns if *none* of the conditions `C1`, `C2`,…, `CN` evaluate to `true` (the `else` clause is optional, which is why it is enclosed in square brackets). All the expressions returned by the various `when` clauses must evaluate to the same type (e.g., `date`, `number`, `varchar`).

Here's an example of a searched case expression:

```
CASE
  WHEN category.name IN
('Children','Family','Sports','Animation')
    THEN 'All Ages'
  WHEN category.name = 'Horror'
    THEN 'Adult'
  WHEN category.name IN ('Music','Games')
    THEN 'Teens'
  ELSE 'Other'
END
```

This case expression returns a string that can be used to classify films depending on their category. When the case expression is evaluated, the `when` clauses are evaluated in order from top to bottom; as soon as one of the conditions in a `when` clause evaluates to `true`, the corresponding expression is returned and any remaining `when` clauses are ignored. If none of the `when` clause conditions evaluate to `true`, then the expression in the `else` clause is returned.

Although the previous example returns string expressions, keep in mind that case expressions may return any type of expression, including subqueries. Here's another version of the query from earlier in the chapter that uses a subquery to return the number of rentals, but only for Active customers:

```
mysql> SELECT c.first_name, c.last_name,
    ->   CASE
    ->     WHEN active = 0 THEN 0
    ->     ELSE
    ->      (SELECT count(*) FROM rental r
```

```
    ->        WHERE r.customer_id = c.customer_id)
    ->   END num_rentals
    -> FROM customer c;
+------------+-------------+-------------+
| first_name | last_name   | num_rentals |
+------------+-------------+-------------+
| MARY       | SMITH       |          32 |
| PATRICIA   | JOHNSON     |          27 |
| LINDA      | WILLIAMS    |          26 |
| BARBARA    | JONES       |          22 |
| ELIZABETH  | BROWN       |          38 |
| JENNIFER   | DAVIS       |          28 |
...
| TERRANCE   | ROUSH       |           0 |
| RENE       | MCALISTER   |          26 |
| EDUARDO    | HIATT       |          27 |
| TERRENCE   | GUNDERSON   |          30 |
| ENRIQUE    | FORSYTHE    |          28 |
| FREDDIE    | DUGGAN      |          25 |
| WADE       | DELVALLE    |          22 |
| AUSTIN     | CINTRON     |          19 |
+------------+-------------+-------------+
599 rows in set (0.01 sec)
```

This version of the query uses a correlated subquery to retrieve the number of rentals for each Active customer. Depending on the percentage of Active customers, using this approach may be more efficient than joining the `customer` and `rental` tables and grouping on the `customer_id` column.

Simple Case Expressions

The *simple case expression* is quite similar to the searched case expression but is a bit less flexible. Here’s the syntax:

```
CASE V0
  WHEN V1 THEN E1
  WHEN V2 THEN E2
  ...
  WHEN VN THEN EN
```

```
  [ELSE ED]
END
```

In the preceding definition, `V0` represents a value, and the symbols `V1`, `V2`,…, `VN` represent values that are to be compared to `V0`. The symbols `E1`, `E2`,…, `EN` represent expressions to be returned by the case expression, and `ED` represents the expression to be returned if none of the values in the set `V1`, `V2`,…, `VN` match the `V0` value.

Here's an example of a simple case expression:

```
CASE category.name
  WHEN 'Children' THEN 'All Ages'
  WHEN 'Family' THEN 'All Ages'
  WHEN 'Sports' THEN 'All Ages'
  WHEN 'Animation' THEN 'All Ages'
  WHEN 'Horror' THEN 'Adult'
  WHEN 'Music' THEN 'Teens'
  WHEN 'Games' THEN 'Teens'
  ELSE 'Other'
END
```

Simple case expressions are less flexible than searched case expressions because you can't specify your own conditions, whereas searched case expressions may include range conditions, inequality conditions, and multipart conditions using `and`/`or`/`not`, so I would recommend using searched case expressions for all but the simplest logic.

Case Expression Examples

The following sections present a variety of examples illustrating the utility of conditional logic in SQL statements.

Result Set Transformations

You may have run into a situation where you are performing aggregations over a finite set of values, such as days of the week, but you want the result set to contain a single row with one column per value instead of one row per value. As an example, let’s say you have been asked to write a query that shows the number of film rentals for May, June, and July of 2005:

```
mysql> SELECT monthname(rental_date) rental_month,
    ->   count(*) num_rentals
    -> FROM rental
    -> WHERE rental_date BETWEEN '2005-05-01' AND '2005-08-
01'
    -> GROUP BY monthname(rental_date);
+--------------+-------------+
| rental_month | num_rentals |
+--------------+-------------+
| May          |        1156 |
| June         |        2311 |
| July         |        6709 |
+--------------+-------------+
3 rows in set (0.01 sec)
```

However, you have also been instructed to return a single row of data with three columns (one for each of the three months). To transform this result set into a single row, you will need to create three columns and, within each column, sum *only* those rows pertaining to the month in question:

```
mysql> SELECT
    ->   SUM(CASE WHEN monthname(rental_date) = 'May' THEN 1
    ->         ELSE 0 END) May_rentals,
    ->   SUM(CASE WHEN monthname(rental_date) = 'June' THEN
1
    ->         ELSE 0 END) June_rentals,
    ->   SUM(CASE WHEN monthname(rental_date) = 'July' THEN
1
    ->         ELSE 0 END) July_rentals
```

```
    -> FROM rental
    -> WHERE rental_date BETWEEN '2005-05-01' AND '2005-08-01';
+-------------+--------------+--------------+
| May_rentals | June_rentals | July_rentals |
+-------------+--------------+--------------+
|        1156 |         2311 |         6709 |
+-------------+--------------+--------------+
1 row in set (0.01 sec)
```

Each of the three columns in the previous query are identical, except for the month value. When the `monthname()` function returns the desired value for that column, the case expression returns the value `1`; otherwise, it returns a `0`. When summed over all rows, each column returns the number of accounts opened for that month. Obviously, such transformations are practical for only a small number of values; generating one column for each year since 1905 would quickly become tedious.

> **NOTE**
>
> Although it is a bit advanced for this book, it is worth pointing out that both SQL Server and Oracle Database 11*g* include `PIVOT` clauses specifically for these types of queries.

Checking for Existence

Sometimes you will want to determine whether a relationship exists between two entities without regard for the quantity. For example, you might want to know whether an actor has appeared in G-rated films, but you only want to know if there was at least one. Here's a query that uses multiple case expressions to generate three output columns, one to show whether the actor has appeared in G-rated films, another for PG-rated films, and a third for NC-17-rated films:

```
mysql> SELECT a.first_name, a.last_name,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM film_actor fa
    ->                    INNER JOIN film f ON fa.film_id =
f.film_id
    ->                  WHERE fa.actor_id = a.actor_id
    ->                    AND f.rating = 'G') THEN 'Y'
    ->     ELSE 'N'
    ->   END g_actor,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM film_actor fa
    ->                    INNER JOIN film f ON fa.film_id =
f.film_id
    ->                  WHERE fa.actor_id = a.actor_id
    ->                    AND f.rating = 'PG') THEN 'Y'
    ->     ELSE 'N'
    ->   END pg_actor,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM film_actor fa
    ->                    INNER JOIN film f ON fa.film_id =
f.film_id
    ->                  WHERE fa.actor_id = a.actor_id
    ->                    AND f.rating = 'NC-17') THEN 'Y'
    ->     ELSE 'N'
    ->   END nc17_actor
    -> FROM actor a
    -> WHERE a.last_name LIKE 'S%' OR a.first_name LIKE
'S%';
+------------+-------------+---------+----------+-----------
-+
| first_name | last_name   | g_actor | pg_actor | nc17_actor
|
+------------+-------------+---------+----------+-----------
-+
| JOE        | SWANK       | Y       | Y        | Y
|
| SANDRA     | KILMER      | Y       | Y        | Y
|
| CAMERON    | STREEP      | Y       | Y        | Y
|
| SANDRA     | PECK        | Y       | Y        | Y
|
| SISSY      | SOBIESKI    | Y       | Y        | N
```

```
|
| NICK        | STALLONE    | Y         | Y          | Y
|
| SEAN        | WILLIAMS    | Y         | Y          | Y
|
| GROUCHO     | SINATRA     | Y         | Y          | Y
|
| SCARLETT    | DAMON       | Y         | Y          | Y
|
| SPENCER     | PECK        | Y         | Y          | Y
|
| SEAN        | GUINESS     | Y         | Y          | Y
|
| SPENCER     | DEPP        | Y         | Y          | Y
|
| SUSAN       | DAVIS       | Y         | Y          | Y
|
| SIDNEY      | CROWE       | Y         | Y          | Y
|
| SYLVESTER   | DERN        | Y         | Y          | Y
|
| SUSAN       | DAVIS       | Y         | Y          | Y
|
| DAN         | STREEP      | Y         | Y          | Y
|
| SALMA       | NOLTE       | Y         | N          | Y
|
| SCARLETT    | BENING      | Y         | Y          | Y
|
| JEFF        | SILVERSTONE | Y         | Y          | Y
|
| JOHN        | SUVARI      | Y         | Y          | Y
|
| JAYNE       | SILVERSTONE | Y         | Y          | Y
|
+------------+-------------+---------+----------+----------
-+
22 rows in set (0.00 sec)
```

Each case expression includes a correlated subquery against the `film_actor` and `film` tables; one looks for films with a G rating, the second for films with a PG rating, and the third for films with a NC-17 rating. Since each `when` clause uses the `exists` operator, the conditions evaluate to `true` as long as the actor has appeared in at least one film with the proper rating.

In other cases, you may care how many rows are encountered, but only up to a point. For example, the next query uses a simple case expression to count the number of copies in inventory for each film, and then returns either `'Out Of Stock'`, `'Scarce'`, `'Available'`, or `'Common'`:

```
mysql> SELECT f.title,
    ->   CASE (SELECT count(*) FROM inventory i
    ->         WHERE i.film_id = f.film_id)
    ->     WHEN 0 THEN 'Out Of Stock'
    ->     WHEN 1 THEN 'Scarce'
    ->     WHEN 2 THEN 'Scarce'
    ->     WHEN 3 THEN 'Available'
    ->     WHEN 4 THEN 'Available'
    ->     ELSE 'Common'
    ->   END film_availability
    -> FROM film f
    -> ;
+-----------------------------+-------------------+
| title                       | film_availability |
+-----------------------------+-------------------+
| ACADEMY DINOSAUR            | Common            |
| ACE GOLDFINGER              | Available         |
| ADAPTATION HOLES            | Available         |
| AFFAIR PREJUDICE            | Common            |
| AFRICAN EGG                 | Available         |
| AGENT TRUMAN                | Common            |
| AIRPLANE SIERRA             | Common            |
| AIRPORT POLLOCK             | Available         |
| ALABAMA DEVIL               | Common            |
| ALADDIN CALENDAR            | Common            |
| ALAMO VIDEOTAPE             | Common            |
| ALASKA PHANTOM              | Common            |
| ALI FOREVER                 | Available         |
| ALICE FANTASIA              | Out Of Stock      |
...
| YOUNG LANGUAGE              | Scarce            |
| YOUTH KICK                  | Scarce            |
| ZHIVAGO CORE                | Scarce            |
```

```
| ZOOLANDER FICTION              | Common             |
| ZORRO ARK                      | Common             |
+------------------------------+--------------------+
1000 rows in set (0.01 sec)
```

For this query, I stopped counting after 5, since every other number greater than 5 will be given the `'Common'` label.

Division-by-Zero Errors

When performing calculations that include division, you should always take care to ensure that the denominators are never equal to zero. Whereas some database servers, such as Oracle Database, will throw an error when a zero denominator is encountered, MySQL simply sets the result of the calculation to `null`, as demonstrated by the following:

```
mysql> SELECT 100 / 0;
+---------+
| 100 / 0 |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)
```

To safeguard your calculations from encountering errors or, even worse, from being mysteriously set to `null`, you should wrap all denominators in conditional logic, as demonstrated by the following:

```
mysql> SELECT c.first_name, c.last_name,
    ->   sum(p.amount) tot_payment_amt,
    ->   count(p.amount) num_payments,
    ->   sum(p.amount) /
    ->     CASE WHEN count(p.amount) = 0 THEN 1
    ->       ELSE count(p.amount)
    ->     END avg_payment
    -> FROM customer c
```

```
    ->   LEFT OUTER JOIN payment p
    ->   ON c.customer_id = p.customer_id
    -> GROUP BY c.first_name, c.last_name;
+------------+-----------+-----------------+--------------
+-------------+
| first_name | last_name  | tot_payment_amt | num_payments |
avg_payment |
+------------+-----------+-----------------+--------------
+-------------+
| MARY       | SMITH      |          118.68 |           32
|    3.708750 |
| PATRICIA   | JOHNSON    |          128.73 |           27
|    4.767778 |
| LINDA      | WILLIAMS   |          135.74 |           26
|    5.220769 |
| BARBARA    | JONES      |           81.78 |           22
|    3.717273 |
| ELIZABETH  | BROWN      |          144.62 |           38
|    3.805789 |
...
| EDUARDO    | HIATT      |          130.73 |           27
|    4.841852 |
| TERRENCE   | GUNDERSON  |          117.70 |           30
|    3.923333 |
| ENRIQUE    | FORSYTHE   |           96.72 |           28
|    3.454286 |
| FREDDIE    | DUGGAN     |           99.75 |           25
|    3.990000 |
| WADE       | DELVALLE   |           83.78 |           22
|    3.808182 |
| AUSTIN     | CINTRON    |           83.81 |           19
|    4.411053 |
+------------+-----------+-----------------+--------------
+-------------+
599 rows in set (0.07 sec)
```

This query computes the average payment amount for each customer. Since some customers may be new and have yet to rent a film, it is best to include the case expression to ensure that the denominator is never zero.

Conditional Updates

When updating rows in a table, you sometimes need conditional logic to generate a value for a column. For example, let's say that you run a job every week that will set the `customer.active` column to 0 for any customers who haven't rented a film in the last 90 days. Here's a statement that will set the value to either 0 or 1 for every customer:

```
UPDATE customer
SET active =
  CASE
    WHEN 90 <= (SELECT datediff(now(), max(rental_date))
                FROM rental r
                WHERE r.customer_id = customer.customer_id)
      THEN 0
    ELSE 1
  END
WHERE active = 1;
```

This statement uses a correlated subquery to determine the number of days since the last rental date for each customer, and compares the value to 90; if the number returned by the subquery is 90 or higher, the customer is marked as Inactive.

Handling Null Values

While `null` values are the appropriate thing to store in a table if the value for a column is unknown, it is not always appropriate to retrieve `null` values for display or to take part in expressions. For example, you might want to display the word *unknown* on a data entry screen rather than leaving a field blank. When retrieving the data, you can use a case expression to substitute the string if the value is `null`, as in:

```
SELECT c.first_name, c.last_name,
  CASE
    WHEN a.address IS NULL THEN 'Unknown'
    ELSE a.address
  END address,
  CASE
    WHEN ct.city IS NULL THEN 'Unknown'
    ELSE ct.city
  END city,
  CASE
    WHEN cn.country IS NULL THEN 'Unknown'
    ELSE cn.country
  END country
FROM customer c
  LEFT OUTER JOIN address a
  ON c.address_id = a.address_id
  LEFT OUTER JOIN city ct
  ON a.city_id = ct.city_id
  LEFT OUTER JOIN country cn
  ON ct.country_id = cn.country_id;
```

For calculations, `null` values often cause a `null` result, as demonstrated by the following:

```
mysql> SELECT (7 * 5) / ((3 + 14) * null);
+-----------------------------+
| (7 * 5) / ((3 + 14) * null) |
+-----------------------------+
|                        NULL |
+-----------------------------+
1 row in set (0.08 sec)
```

When performing calculations, case expressions are useful for translating a `null` value into a number (usually 0 or 1) that will allow the calculation to yield a non-`null` value.

Test Your Knowledge

Challenge your ability to work through conditional logic problems with the examples that follow. When you're done, compare your solutions with those in Appendix C.

Exercise 11-1

Rewrite the following query, which uses a simple case expression, so that the same results are achieved using a searched case expression. Try to use as few `when` clauses as possible.

```
SELECT name,
  CASE name
    WHEN 'English' THEN 'latin1'
    WHEN 'Italian' THEN 'latin1'
    WHEN 'French' THEN 'latin1'
    WHEN 'German' THEN 'latin1'
    WHEN 'Japanese' THEN 'utf8'
    WHEN 'Mandarin' THEN 'utf8'
    ELSE 'Unknown'
  END character_set
FROM language;
```

Exercise 11-2

Rewrite the following query so that the result set contains a single row with five columns (one for each rating). Name the five columns G, PG, PG_13, R, and NC_17.

```
mysql> SELECT rating, count(*)
    -> FROM film
    -> GROUP BY rating;
+--------+----------+
| rating | count(*) |
+--------+----------+
| PG     |      194 |
| G      |      178 |
| NC-17  |      210 |
| PG-13  |      223 |
| R      |      195 |
```

```
+--------+----------+
5 rows in set (0.00 sec)
```

Chapter 12. Transactions

All of the examples thus far in this book have been individual, independent SQL statements. While this may be the norm for ad hoc reporting or data maintenance scripts, application logic will frequently include multiple SQL statements that need to execute together as a logical unit of work. This chapter explores the need and the infrastructure necessary to execute multiple SQL statements concurrently.

Multiuser Databases

Database management systems allow not only a single user to query and modify data, but multiple people to do so simultaneously. If every user is only executing queries, such as might be the case with a data warehouse during normal business hours, then there are very few issues for the database server to deal with. If some of the users are adding and/or modifying data, however, the server must handle quite a bit more bookkeeping.

Let's say, for example, that you are running a report that sums up the current week's film rental activity. At the same time you are running the report, however, the following activities are occurring:

- A customer rents a film.

- A customer returns a film after the due date and pays a late fee.
- Five new films are added to inventory.

While your report is running, therefore, multiple users are modifying the underlying data, so what numbers should appear on the report? The answer depends somewhat on how your server handles *locking*, which is described in the next section.

Locking

Locks are the mechanism the database server uses to control simultaneous use of data resources. When some portion of the database is locked, any other users wishing to modify (or possibly read) that data must wait until the lock has been released. Most database servers use one of two locking strategies:

- Database writers must request and receive from the server a *write lock* to modify data, and database readers must request and receive from the server a *read lock* to query data. While multiple users can read data simultaneously, only one write lock is given out at a time for each table (or portion thereof), and read requests are blocked until the write lock is released.
- Database writers must request and receive from the server a write lock to modify data, but readers do not need any type of lock to query data. Instead, the server ensures that a reader sees a consistent view of the data (the data seems the same even though other users may be making modifications) from the time her query begins until her query has finished. This approach is known as *versioning*.

There are pros and cons to both approaches. The first approach can lead to long wait times if there are many concurrent read and write requests, and the second approach can be problematic if there are long-running queries while data is being modified. Of the three servers discussed in this book, Microsoft SQL Server uses the first approach, Oracle Database uses the second approach, and MySQL uses both approaches (depending on your choice of *storage engine*, which we'll discuss a bit later in the chapter).

Lock Granularities

There are also a number of different strategies that you may employ when deciding *how* to lock a resource. The server may apply a lock at one of three different levels, or *granularities*:

Table locks

Keep multiple users from modifying data in the same table simultaneously

Page locks

Keep multiple users from modifying data on the same page (a page is a segment of memory generally in the range of 2 KB to 16 KB) of a table simultaneously

Row locks

Keep multiple users from modifying the same row in a table simultaneously

Again, there are pros and cons to these approaches. It takes very little bookkeeping to lock entire tables, but this approach quickly yields unacceptable wait times as the number of users increases. On the other hand, row locking takes quite a bit more bookkeeping, but it allows many users to modify the same table as long as they are interested in different rows. Of the three servers discussed in this book, Microsoft SQL Server uses page, row, and table locking, Oracle Database uses only row locking, and MySQL uses table, page, or row locking (depending, again, on your choice of storage engine). SQL Server will, under certain circumstances, *escalate* locks from row to page, and from page to table, whereas Oracle Database will never escalate locks.

To get back to your report, the data that appears on the pages of the report will mirror either the state of the database when your report started (if your server uses a versioning approach) or the state of the database when the server issues the reporting application a read lock (if your server uses both read and write locks).

What Is a Transaction?

If database servers enjoyed 100% uptime, if users always allowed programs to finish executing, and if applications always completed without encountering fatal errors that halt execution, then there would be nothing left to discuss regarding concurrent database access. However, we can rely on none of these things, so one more element is necessary to allow multiple users to access the same data.

This extra piece of the concurrency puzzle is the *transaction*, which is a device for grouping together multiple SQL statements such that either *all* or *none* of the statements succeed (a property known as *atomicity*). If you attempt to transfer $500 from your savings account to your checking account, you would be a bit upset if the money were successfully withdrawn from your savings account but never made it to your checking account. Whatever the reason for the failure (the server was shut down for maintenance, the request for a page lock on the `account` table timed out, etc.), you want your $500 back.

To protect against this kind of error, the program that handles your transfer request would first begin a transaction, then issue the SQL statements needed to move the money from your savings to your checking account, and, if everything succeeds, end the transaction by issuing the `commit` command. If something unexpected happens, however, the program would issue a `rollback` command, which instructs the server to undo all changes made since the transaction began. The entire process might look something like the following:

```
START TRANSACTION;

 /* withdraw money from first account, making sure balance
is sufficient */
UPDATE account SET avail_balance = avail_balance - 500
WHERE account_id = 9988
  AND avail_balance > 500;

IF <exactly one row was updated by the previous statement>
THEN
  /* deposit money into second account */
  UPDATE account SET avail_balance = avail_balance + 500
    WHERE account_id = 9989;

  IF <exactly one row was updated by the previous statement>
THEN
    /* everything worked, make the changes permanent */
    COMMIT;
  ELSE
    /* something went wrong, undo all changes in this
transaction */
    ROLLBACK;
  END IF;
```

```
ELSE
  /* insufficient funds, or error encountered during update
*/
  ROLLBACK;
END IF;
```

> **NOTE**
>
> While the previous code block may look similar to one of the procedural languages provided by the major database companies, such as Oracle's PL/SQL or Microsoft's Transact-SQL, it is written in pseudocode and does not attempt to mimic any particular language.

The previous code block begins by starting a transaction and then attempts to remove $500 from the checking account and add it to the savings account. If all goes well, the transaction is committed; if anything goes awry, however, the transaction is rolled back, meaning that all data changes since the beginning of the transaction are undone.

By using a transaction, the program ensures that your $500 either stays in your savings account or moves to your checking account, without the possibility of it falling into a crack. Regardless of whether the transaction was committed or was rolled back, all resources acquired (e.g., write locks) during the execution of the transaction are released when the transaction completes.

Of course, if the program manages to complete both `update` statements but the server shuts down before a `commit` or `rollback` can be executed, then the transaction will be rolled back when the server comes back online. (One of the tasks that a database server must complete before coming online is to find any incomplete transactions that were underway when the server shut down and roll them back.) Additionally, if your program finishes a transaction and issues a `commit`, but the server shuts down before the changes have been applied to permanent storage (i.e., the modified data is sitting in memory but has not been flushed to disk), then the database server must reapply the changes from your transaction when the server is restarted (a property known as durability).

Starting a Transaction

Database servers handle transaction creation in one of two ways:

- An active transaction is always associated with a database session, so there is no need or method to explicitly begin a transaction. When the current transaction ends, the server automatically begins a new transaction for your session.
- Unless you explicitly begin a transaction, individual SQL statements are automatically committed independently of one another. To begin a transaction, you must first issue a command.

Of the three servers, Oracle Database takes the first approach, while Microsoft SQL Server and MySQL take the second approach. One of the advantages of Oracle's approach to transactions is that, even if you are issuing only a single SQL command, you have the ability to roll back the changes if you don't like the outcome or if you change your mind. Thus, if you forget to add a `where` clause to your `delete` statement, you will have the opportunity to undo the damage (assuming you've had your morning coffee and realize that you didn’t mean to delete all 125,000 rows in your table). With MySQL and SQL Server, however, once you press the Enter key, the changes brought about by your SQL statement will be permanent (unless your DBA can retrieve the original data from a backup or from some other means).

The SQL:2003 standard includes a `start transaction` command to be used when you want to explicitly begin a transaction. While MySQL conforms to the standard, SQL Server users must instead issue the command `begin transaction`. With both servers, until you explicitly begin a transaction, you are in what is known as *auto-commit mode*, which means that individual statements are automatically committed by the server. You can, therefore, decide that you want to be in a transaction and issue a start/begin transaction command, or you can simply let the server commit individual statements.

Both MySQL and SQL Server allow you to turn off auto-commit mode for individual sessions, in which case, the servers will act just like Oracle Database regarding transactions. With SQL Server, you issue the following command to disable auto-commit mode:

```
SET IMPLICIT_TRANSACTIONS ON
```

MySQL allows you to disable auto-commit mode via the following:

```
SET AUTOCOMMIT=0
```

Once you have left auto-commit mode, all SQL commands take place within the scope of a transaction and must be explicitly committed or rolled back.

> **NOTE**
>
> A word of advice: shut off auto-commit mode each time you log in, and get in the habit of running all of your SQL statements within a transaction. If nothing else, it may save you the embarrassment of having to ask your DBA to reconstruct data that you have inadvertently deleted.

Ending a Transaction

Once a transaction has begun, whether explicitly via the `start transaction` command or implicitly by the database server, you must explicitly end your transaction for your changes to become permanent. You do this by way of the `commit` command, which instructs the server to mark the changes as permanent and release any resources (i.e., page or row locks) used during the transaction.

If you decide that you want to undo all the changes made since starting the transaction, you must issue the `rollback` command, which instructs the server to return the data to its pre-transaction state. After the `rollback` has been completed, any resources used by your session are released.

Along with issuing either the `commit` or `rollback` command, there are several other scenarios by which your transaction can end, either as an indirect result of your actions or as a result of something outside your control:

- The server shuts down, in which case, your transaction will be rolled back automatically when the server is restarted.
- You issue an SQL schema statement, such as `alter table`, which will cause the current transaction to be committed and a new transaction to be started.
- You issue another `start transaction` command, which will cause the previous transaction to be committed.
- The server prematurely ends your transaction because the server detects a *deadlock* and decides that your transaction is the culprit. In this case, the transaction will be rolled back and you will receive an error message.

Of these four scenarios, the first and third are fairly straightforward, but the other two merit some discussion. As far as the second scenario is concerned, alterations to a database, whether it be the addition of a new table or index or the removal of a column from a table, cannot be rolled back, so commands that alter your schema must take place outside a transaction. If a transaction is currently underway, therefore, the server will commit your current transaction, execute the SQL schema statement command(s), and then automatically start a new transaction for your session. The server will not inform you of what has happened, so you should be careful that the statements that comprise a unit of work are not inadvertently broken up into multiple transactions by the server.

The fourth scenario deals with deadlock detection. A deadlock occurs when two different transactions are waiting for resources that the other transaction currently holds. For example, transaction A might have just updated the `account` table and is waiting for a write lock on the `transaction` table, while transaction B has inserted a row into the `transaction` table and is waiting for a write lock on the `account` table. If both transactions happen to be modifying the same page or row (depending on the lock granularity in use by the database server), then they will each wait forever for the other transaction to finish and free up the needed resource. Database servers must always be on the lookout for these situations so that throughput doesn't grind to a halt; when a deadlock is detected, one of the transactions is chosen (either arbitrarily or by some criteria) to be rolled back so that the other transaction may proceed. Most of the time, the terminated transaction can be restarted and will succeed without encountering another deadlock situation.

Unlike the second scenario discussed earlier, the database server will raise an error to inform you that your transaction has been rolled back due to deadlock detection. With MySQL, for example, you will receive error #1213, which carries the following message:

```
Message: Deadlock found when trying to get lock; try
restarting transaction
```

As the error message suggests, it is a reasonable practice to retry a transaction that has been rolled back due to deadlock detection. However, if deadlocks become fairly common, then you may need to modify the applications that access the database to decrease the probability of deadlocks (one common strategy is to ensure that data resources are always accessed in the same order, such as always modifying account data before inserting transaction data).

Transaction Savepoints

In some cases, you may encounter an issue within a transaction that requires a rollback, but you may not want to undo *all* of the work that has transpired. For these situations, you can establish one or more *savepoints* within a transaction and use them to roll back to a particular location within your transaction rather than rolling all the way back to the start of the transaction.

CHOOSING A STORAGE ENGINE

When using Oracle Database or Microsoft SQL Server, a single set of code is responsible for low-level database operations, such as retrieving a particular row from a table based on primary key value. The MySQL server, however, has been designed so that multiple storage engines may be utilized to provide low-level database functionality, including resource locking and transaction management. As of version 8.0, MySQL includes the following storage engines:

MyISAM

A nontransactional engine employing table locking

MEMORY

A nontransactional engine used for in-memory tables

CSV

A transactional engine which stores data in comma-separated files

InnoDB

A transactional engine employing row-level locking

Merge

A specialty engine used to make multiple identical MyISAM tables appear as a single table (a.k.a. table partitioning)

Archive

A specialty engine used to store large amounts of unindexed data, mainly for archival purposes

Although you might think that you would be forced to choose a single storage engine for your database, MySQL is flexible enough to allow you to choose a storage engine on a table-by-table basis. For any tables that might take part in transactions, however, you should choose the InnoDB engine, which uses row-level locking and versioning to provide the highest level of concurrency across the different storage engines.

You may explicitly specify a storage engine when creating a table, or you can change an existing table to use a different engine. If you do not know what engine is assigned to a table, you can use the `show table` command, as demonstrated by the following:

```
mysql> show table status like 'customer' \G;
*************************** 1. row ***************************
           Name: customer
         Engine: InnoDB
        Version: 10
     Row_format: Dynamic
           Rows: 599
 Avg_row_length: 136
    Data_length: 81920
Max_data_length: 0
```

```
    Index_length: 49152
       Data_free: 0
  Auto_increment: 599
     Create_time: 2019-03-12 14:24:46
     Update_time: NULL
      Check_time: NULL
       Collation: utf8_general_ci
        Checksum: NULL
  Create_options:
         Comment:
1 row in set (0.16 sec)
```

Looking at the second item, you can see that the Customer table is already using the InnoDB engine. If it were not, you could assign the InnoDB engine to the transaction table via the following command:

```
ALTER TABLE customer ENGINE = INNODB;
```

All savepoints must be given a name, which allows you to have multiple savepoints within a single transaction. To create a savepoint named `my_savepoint`, you can do the following:

```
SAVEPOINT my_savepoint;
```

To roll back to a particular savepoint, you simply issue the `rollback` command followed by the keywords `to savepoint` and the name of the savepoint, as in:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Here's an example of how savepoints may be used:

```
START TRANSACTION;

UPDATE product
SET date_retired = CURRENT_TIMESTAMP()
WHERE product_cd = 'XYZ';

SAVEPOINT before_close_accounts;

UPDATE account
SET status = 'CLOSED', close_date = CURRENT_TIMESTAMP(),
```

```
  last_activity_date = CURRENT_TIMESTAMP()
WHERE product_cd = 'XYZ';

ROLLBACK TO SAVEPOINT before_close_accounts;
COMMIT;
```

The net effect of this transaction is that the mythical XYZ product is retired but none of the accounts are closed.

When using savepoints, remember the following:

- Despite the name, nothing is saved when you create a savepoint. You must eventually issue a `commit` if you want your transaction to be made permanent.
- If you issue a `rollback` without naming a savepoint, all savepoints within the transaction will be ignored and the entire transaction will be undone.

If you are using SQL Server, you will need to use the proprietary command `save transaction` to create a savepoint and `rollback transaction` to roll back to a savepoint, with each command being followed by the savepoint name.

Test Your Knowledge

Test your understanding of transactions by working through the following exercise. When you're done, compare your solutions with Appendix C.

Exercise 12-1

Generate a unit of work to transfer $50 from account 123 to account 789. You will need to insert two rows into the `transaction` table and update two rows in the `account` table. Use the following table definitions/data:

```
                        Account:
account_id      avail_balance   last_activity_date
----------      -------------   ------------------
123             500             2019-07-10 20:53:27
789             75              2019-06-22 15:18:35

                        Transaction:
txn_id          txn_date        account_id      txn_type_cd
amount
---------       ------------    -----------     -----------
--------
1001            2019-05-15      123             C
500
1002            2019-06-01      789             C
75
```

Chapter 13. Indexes and Constraints

Because the focus of this book is on programming techniques, the first 12 chapters concentrated on elements of the SQL language that you can use to craft powerful `select`, `insert`, `update`, and `delete` statements. However, other database features *indirectly* affect the code you write. This chapter focuses on two of those features: indexes and constraints.

Indexes

When you insert a row into a table, the database server does not attempt to put the data in any particular location within the table. For example, if you add a row to the `customer` table, the server doesn't place the row in numeric order via the `customer_id` column or in alphabetical order via the `last_name` column. Instead, the server simply places the data in the next available location within the file (the server maintains a list of free space for each table). When you query the `customer` table, therefore, the server will need to inspect every row of the table to answer the query. For example, let's say that you issue the following query:

```
mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE last_name LIKE 'Y%';
+------------+-----------+
```

```
| first_name | last_name |
+------------+-----------+
| LUIS       | YANEZ     |
| MARVIN     | YEE       |
| CYNTHIA    | YOUNG     |
+------------+-----------+
3 rows in set (0.09 sec)
```

To find all customers whose last name begins with *Y*, the server must visit each row in the `customer` table and inspect the contents of the `last_name` column; if the department name begins with *Y*, then the row is added to the result set. This type of access is known as a *table scan*.

While this method works fine for a table with only three rows, imagine how long it might take to answer the query if the table contains 3 million rows. At some number of rows larger than three and smaller than 3 million, a line is crossed where the server cannot answer the query within a reasonable amount of time without additional help. This help comes in the form of one or more *indexes* on the `customer` table.

Even if you have never heard of a database index, you are certainly aware of what an index is (e.g., this book has one). An index is simply a mechanism for finding a specific item within a resource. Each technical publication, for example, includes an index at the end that allows you to locate a specific word or phrase within the publication. The index lists these words and phrases in alphabetical order, allowing the reader to move quickly to a particular letter within the index, find the desired entry, and then find the page or pages on which the word or phrase may be found.

In the same way that a person uses an index to find words within a publication, a database server uses indexes to locate rows in a table. Indexes are special tables that, unlike normal data tables, *are* kept in a specific order. Instead of containing *all* of the data about an entity, however, an index contains only the column (or columns) used to locate rows in the data table, along with information describing where the rows are physically located. Therefore, the role of indexes is to facilitate the retrieval of a subset of a table's rows and columns *without* the need to inspect every row in the table.

Index Creation

Returning to the `customer` table, you might decide to add an index on the `email` column to speed up any queries that specify a value for this column, as well as any `update` or `delete` operations that specify a customer's email address. Here's how you can add such an index to a MySQL database:

```
mysql> ALTER TABLE customer
    -> ADD INDEX idx_email (email);
Query OK, 0 rows affected (1.87 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This statement creates an index (a B-tree index to be precise, but more on this shortly) on the `customer.email` column; furthermore, the index is given the name `idx_email`. With the index in place, the query optimizer (which we discussed in Chapter 3) can choose to use the index if it is deemed beneficial to do so. If there is more than one index on a table, the optimizer must decide which index will be the most beneficial for a particular SQL statement.

NOTE

MySQL treats indexes as optional components of a table, which is why in earlier versions you would use the `alter table` command to add or remove an index. Other database servers, including SQL Server and Oracle Database, treat indexes as independent schema objects. For both SQL Server and Oracle, therefore, you would generate an index using the `create index` command, as in:

```
CREATE INDEX idx_email
ON customer (email);
```

As of MySQL version 5, a `create index` command command is available, although it is mapped to the `alter table` command. You must still use the `alter table` command to create primary key indexes, however.

All database servers allow you to look at the available indexes. MySQL users can use the `show` command to see all of the indexes on a specific table, as in:

```
mysql> SHOW INDEX FROM customer \G;
*************************** 1. row
***************************
        Table: customer
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: customer_id
    Collation: A
  Cardinality: 599
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
...
*************************** 2. row
***************************
        Table: customer
   Non_unique: 1
     Key_name: idx_fk_store_id
```

```
 Seq_in_index: 1
  Column_name: store_id
    Collation: A
  Cardinality: 2
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
...
*************************** 3. row
***************************
        Table: customer
   Non_unique: 1
     Key_name: idx_fk_address_id
 Seq_in_index: 1
  Column_name: address_id
    Collation: A
  Cardinality: 599
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
...
*************************** 4. row
***************************
        Table: customer
   Non_unique: 1
     Key_name: idx_last_name
 Seq_in_index: 1
  Column_name: last_name
    Collation: A
  Cardinality: 599
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
...
*************************** 5. row
***************************
        Table: customer
   Non_unique: 1
     Key_name: idx_email
 Seq_in_index: 1
  Column_name: email
    Collation: A
  Cardinality: 599
     Sub_part: NULL
       Packed: NULL
```

```
         Null: YES
   Index_type: BTREE
...
5 rows in set (0.06 sec)
```

The output shows that there are five indexes on the `customer` table: one on the `customer_id` column called `PRIMARY`, and four others on the `store_id`, `address_id`, `last_name`, and `email` columns. If you are wondering where these indexes came from, I created the index on the `email` column, and the rest were installed as part of the sample Sakila database. Here's the statement used to create the table:

```
CREATE TABLE customer (
  customer_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  store_id TINYINT UNSIGNED NOT NULL,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  email VARCHAR(50) DEFAULT NULL,
  address_id SMALLINT UNSIGNED NOT NULL,
  active BOOLEAN NOT NULL DEFAULT TRUE,
  create_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY  (customer_id),
  KEY idx_fk_store_id (store_id),
  KEY idx_fk_address_id (address_id),
  KEY idx_last_name (last_name),
  ...
```

When the table was created, the MySQL server automatically generated an index on the primary key column, which, in this case is `customer_id`, and gave the index the name `PRIMARY`. I cover constraints later in this chapter.

If, after creating an index, you decide that the index is not proving useful, you can remove it via the following:

```
mysql> ALTER TABLE customer
    -> DROP INDEX idx_email;
Query OK, 0 rows affected (0.50 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

> **NOTE**
>
> SQL Server and Oracle Database users must use the `drop index` command to remove an index, as in:
>
> ```
> DROP INDEX idx_email; (Oracle)
>
> DROP INDEX idx_email ON customer; (SQL Server)
> ```
>
> MySQL now also supports the `drop index` command, although it is also mapped to the `alter table` command.

UNIQUE INDEXES

When designing a database, it is important to consider which columns are allowed to contain duplicate data and which are not. For example, it is allowable to have two customers named John Smith in the `customer` table since each row will have a different identifier (`customer_id`), email, and address to help tell them apart. You would not, however, want to allow two different customers to have the same email address. You can enforce a rule against duplicate values by creating a *unique index* on the `customer.email` column.

A unique index plays multiple roles; along with providing all the benefits of a regular index, it also serves as a mechanism for disallowing duplicate values in the indexed column. Whenever a row is inserted or when the indexed column is modified, the database server checks the unique index to see whether the value already exists in another row in the table. Here’s how you would create a unique index on the `customer.email` column:

```
mysql> ALTER TABLE customer
    -> ADD UNIQUE idx_email (email);
Query OK, 0 rows affected (0.64 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

NOTE

SQL Server and Oracle Database users need only add the `unique` keyword when creating an index, as in:

```
CREATE UNIQUE INDEX idx_email
ON customer (email);
```

With the index in place, you will receive an error if you try to add a new customer with an email address which already exists:

```
mysql> INSERT INTO customer
    ->  (store_id, first_name, last_name, email, address_id,
active)
    -> VALUES
    ->  (1,'ALAN','KAHN', 'ALAN.KAHN@sakilacustomer.org',
394, 1);
ERROR 1062 (23000): Duplicate entry
'ALAN.KAHN@sakilacustomer.org'
  for key 'idx_email'
```

You should not build unique indexes on your primary key column(s), since the server already checks uniqueness for primary key values. You may, however, create more than one unique index on the same table if you feel that it is warranted.

MULTICOLUMN INDEXES

Along with the single-column indexes demonstrated thus far, you may also build indexes that span multiple columns. If, for example, you find yourself searching for customers by first *and* last names, you can build an index on *both* columns together, as in:

```
mysql> ALTER TABLE customer
    -> ADD INDEX idx_full_name (last_name, first_name);
Query OK, 0 rows affected (0.35 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This index will be useful for queries that specify the first and last names or just the last name, but it would not be useful for queries that specify only the customer's first name. To understand why, consider how you would find a person's phone number; if you know the person's first and last names, you can use a phone book to find the number quickly, since a phone book is organized by last name and then by first name. If you know only the person's first name, you would need to scan every entry in the phone book to find all the entries with the specified first name.

When building multiple-column indexes, therefore, you should think carefully about which column to list first, which column to list second, and so on to help make the index as useful as possible. Keep in mind, however, that there is nothing stopping you from building multiple indexes using the same set of columns but in a different order if you feel that it is needed to ensure adequate response time.

Types of Indexes

Indexing is a powerful tool, but since there are many different types of data, a single indexing strategy doesn't always do the job. The following sections illustrate the different types of indexing available from various servers.

B-TREE INDEXES

All the indexes shown thus far are *balanced-tree indexes*, which are more commonly known as *B-tree indexes*. MySQL, Oracle Database, and SQL Server all default to B-tree indexing, so you will get a B-tree index unless you explicitly ask for another type. As you might expect, B-tree indexes are organized as trees, with one or more levels of *branch nodes* leading to a single level of *leaf nodes*. Branch nodes are used for navigating the tree, while leaf nodes hold the actual values and location information. For example, a B-tree index built on the `customer.last_name` column might look something like Figure 13-1.

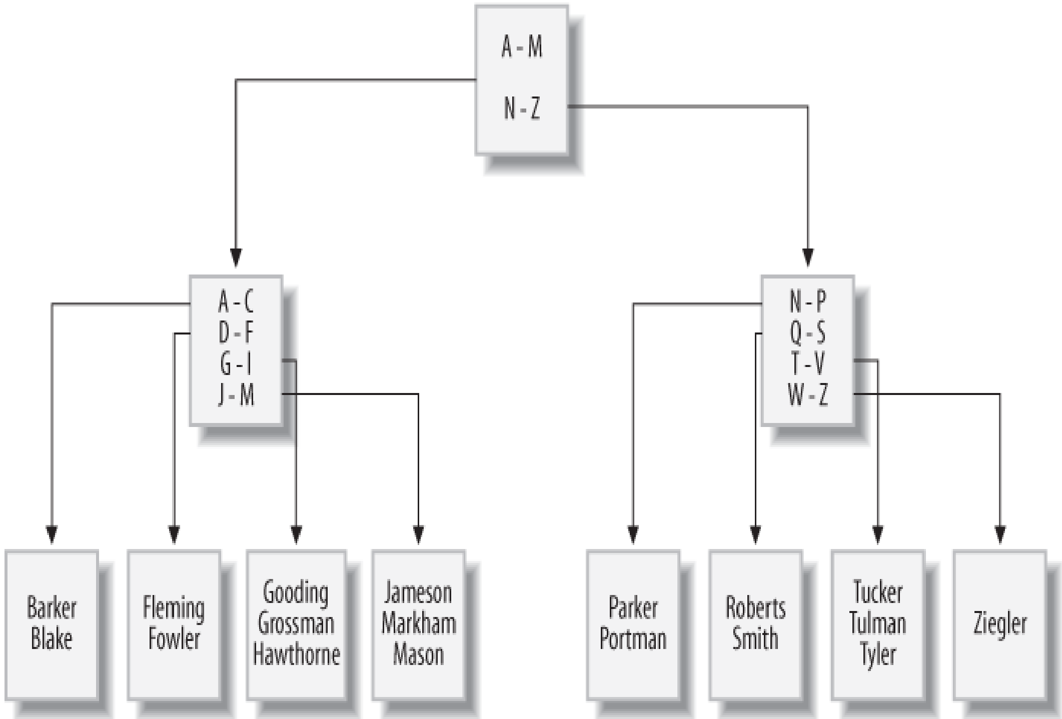

Figure 13-1. B-tree example

If you were to issue a query to retrieve all customers whose last name starts with *G*, the server would look at the top branch node (called the *root node*) and follow the link to the branch node that handles last names beginning with *A* through *M*. This branch node would, in turn, direct the server to a leaf node containing last names beginning with *G* through *I*. The server then starts reading the values in the leaf node until it encounters a value that doesn't begin with *G* (which, in this case, is Hawthorne).

As rows are inserted, updated, and deleted from the `customer` table, the server will attempt to keep the tree balanced so that there aren't far more branch/leaf nodes on one side of the root node than the other. The server can add or remove branch nodes to redistribute the values more evenly and can even add or remove an entire level of branch nodes. By keeping the tree balanced, the server is able to traverse quickly to the leaf nodes to find the desired values without having to navigate through many levels of branch nodes.

BITMAP INDEXES

Although B-tree indexes are great at handling columns that contain many different values, such as a customer's first/last names, they can become unwieldy when built on a column that allows only a small number of values. For example, you may decide to generate an index on the `customer.active` column so that you can quickly retrieve all Active or Inactive accounts. Because there are only two different values (stored as 1 for Active and 0 for Inactive), however, and because there are far more Active customers, it can be difficult to maintain a balanced B-tree index as the number of customers grows.

For columns that contain only a small number of values across a large number of rows (known as *low-cardinality* data), a different indexing strategy is needed. To handle this situation more efficiently, Oracle Database includes *bitmap indexes*, which generate a bitmap for each value stored in the column. If you were to build a bitmap index on the `customer.active` column, the index would maintain two bitmaps: one for the value 0, and another for the value 1. When you write a query to retrieve all Inactive customers, the database server can traverse the 0 bitmap and quickly retrieve the desired rows.

Bitmap indexes are a nice, compact indexing solution for low-cardinality data, but this indexing strategy breaks down if the number of values stored in the column climbs too high in relation to the number of rows (known as *high-cardinality* data), since the server would need to maintain too many bitmaps. For example, you would never build a bitmap index on your primary key column, since this represents the highest possible cardinality (a different value for every row).

Oracle users can generate bitmap indexes by simply adding the `bitmap` keyword to the `create index` statement, as in:

```
CREATE BITMAP INDEX idx_active ON customer (active);
```

Bitmap indexes are commonly used in data warehousing environments, where large amounts of data are generally indexed on columns containing relatively few values (e.g., sales quarters, geographic regions, products, salespeople).

TEXT INDEXES

If your database stores documents, you may need to allow users to search for words or phrases in the documents. You certainly don't want the server to peruse each document and scan for the desired text each time a search is requested, but traditional indexing strategies don't work for this situation. To handle this situation, MySQL, SQL Server, and Oracle Database include specialized indexing and search mechanisms for documents; both SQL Server and MySQL include what they call *full-text* indexes, and Oracle Database includes a powerful set of tools known as *Oracle Text*. Document searches are specialized enough that I refrain from showing an example, but I wanted you to at least know what is available.

How Indexes Are Used

Indexes are generally used by the server to quickly locate rows in a particular table, after which the server visits the associated table to extract the additional information requested by the user. Consider the following query:

```
mysql> SELECT customer_id, first_name, last_name
    -> FROM customer
    -> WHERE first_name LIKE 'S%' AND last_name LIKE 'P%';
+-------------+------------+-----------+
| customer_id | first_name | last_name |
+-------------+------------+-----------+
|          84 | SARA       | PERRY     |
|         197 | SUE        | PETERS    |
|         167 | SALLY      | PIERCE    |
+-------------+------------+-----------+
3 rows in set (0.00 sec)
```

For this query, the server can employ any of the following strategies:

- Scan all rows in the `customer` table
- Use the index on the `last_name` column to find all customers whose last name starts with P, and then visit each row of the `customer` table to find only rows whose first name starts with S
- Use the index on the `last_name, first_name` columns to find all customers whose last name starts with P and whose first name starts with S

The third choice seems to be the best option, since the index will yield all of the rows needed for the result set, without the need to revisit the table. But how do you know which of the three options will be utilized? To see how MySQL's query optimizer decides to execute the query, I use the `explain` statement to ask the server to show the execution plan for the query rather than executing the query:

```
mysql> EXPLAIN
    -> SELECT customer_id, first_name, last_name
    -> FROM customer
    -> WHERE first_name LIKE 'S%' AND last_name LIKE 'P%'
\G;
*************************** 1. row
***************************
           id: 1
  select_type: SIMPLE
        table: customer
   partitions: NULL
         type: range
possible_keys: idx_last_name,idx_full_name
          key: idx_full_name
      key_len: 274
          ref: NULL
         rows: 28
     filtered: 11.11
        Extra: Using where; Using index
1 row in set, 1 warning (0.00 sec)
```

NOTE

Each database server includes tools to allow you to see how the query optimizer handles your SQL statement. SQL Server allows you to see an execution plan by issuing the statement `set showplan_text on` before running your SQL statement. Oracle Database includes the `explain plan` statement, which writes the execution plan to a special table called `plan_table`.

Looking at the query results, the `possible_keys` column tells you that the server could decide to use either the `idx_last_name` or the `idx_full_name` index, and the `key` column tells you that the `idx_full_name` index was chosen. Furthermore, the `type` column tells you that a range-scan will be utilized, meaning that the database server will be looking for a range of values in the index, rather than expecting to retrieve a single row.

> **NOTE**
>
> The process that I just led you through is an example of query tuning. Tuning involves looking at an SQL statement and determining the resources available to the server to execute the statement. You can decide to modify the SQL statement, to adjust the database resources, or to do both in order to make a statement run more efficiently. Tuning is a detailed topic, and I strongly urge you to either read your server's tuning guide or pick up a good tuning book so that you can see all the different approaches available for your server.

The Downside of Indexes

If indexes are so great, why not index everything? Well, the key to understanding why more indexes are not necessarily a good thing is to keep in mind that every index is a table (a special type of table, but still a table). Therefore, every time a row is added to or removed from a table, all indexes on that table must be modified. When a row is updated, any indexes on the column or columns that were affected need to be modified as well. Therefore, the more indexes you have, the more work the server needs to do to keep all schema objects up-to-date, which tends to slow things down.

Indexes also require disk space as well as some amount of care from your administrators, so the best strategy is to add an index when a clear need arises. If you need an index for only special purposes, such as a monthly maintenance routine, you can always add the index, run the routine, and then drop the index until you need it again. In the case of data warehouses, where indexes are crucial during business hours as users run reports and ad hoc queries but are problematic when data is being loaded into the warehouse overnight, it is a common practice to drop the indexes before data is loaded and then re-create them before the warehouse opens for business.

In general, you should strive to have neither too many indexes nor too few. If you aren’t sure how many indexes you should have, you can use this strategy as a default:

- Make sure all primary key columns are indexed (most servers automatically create unique indexes when you create primary key constraints). For multicolumn primary keys, consider building additional indexes on a subset of the primary key columns, or on all the primary key columns but in a different order than the primary key constraint definition.
- Build indexes on all columns that are referenced in foreign key constraints. Keep in mind that the server checks to make sure there are no child rows when a parent is deleted, so it must issue a query to search for a particular value in the column. If there’s no index on the column, the entire table must be scanned.
- Index any columns that will frequently be used to retrieve data. Most date columns are good candidates, along with short (2- to 50-character) string columns.

After you have built your initial set of indexes, try to capture actual queries against your tables, look at the server’s execution plan, and modify your indexing strategy to fit the most-common access paths.

Constraints

A constraint is simply a restriction placed on one or more columns of a table. There are several different types of constraints, including:

Primary key constraints

> Identify the column or columns that guarantee uniqueness within a table

Foreign key constraints

> Restrict one or more columns to contain only values found in another table's primary key columns, and may also restrict the allowable values in other tables if `update cascade` or `delete cascade` rules are established

Unique constraints

> Restrict one or more columns to contain unique values within a table (primary key constraints are a special type of unique constraint)

Check constraints

> Restrict the allowable values for a column

Without constraints, a database's consistency is suspect. For example, if the server allows you to change a customer's ID in the `customer` table without changing the same customer ID in the `rental` table, then you will end up with rental data that no longer point to valid customer records (known as *orphaned rows*). With primary and foreign key constraints in place, however, the server will either raise an error if an attempt is made to modify or delete data that is referenced by other tables, or propagate the changes to other tables for you (more on this shortly).

> **NOTE**
>
> If you want to use foreign key constraints with the MySQL server, you must use the InnoDB storage engine for your tables.

Constraint Creation

Constraints are generally created at the same time as the associated table via the `create table` statement. To illustrate, here's an example from the schema generation script for the Sakila sample database:

```
CREATE TABLE customer (
  customer_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  store_id TINYINT UNSIGNED NOT NULL,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  email VARCHAR(50) DEFAULT NULL,
  address_id SMALLINT UNSIGNED NOT NULL,
  active BOOLEAN NOT NULL DEFAULT TRUE,
  create_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (customer_id),
  KEY idx_fk_store_id (store_id),
  KEY idx_fk_address_id (address_id),
  KEY idx_last_name (last_name),
  CONSTRAINT fk_customer_address FOREIGN KEY (address_id)
    REFERENCES address (address_id) ON DELETE RESTRICT ON
UPDATE CASCADE,
  CONSTRAINT fk_customer_store FOREIGN KEY (store_id)
    REFERENCES store (store_id) ON DELETE RESTRICT ON UPDATE
CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `customer` table includes three constraints: one to specify that the `customer_id` column serves as the primary key for the table, and two more to specify that the `address_id` and `store_id` columns serve as foreign keys to the `address` and `store` table. Alternatively, you could create the `customer` table without foreign key constraints, and add the foreign key constraints later via `alter table` statements:

```
ALTER TABLE customer
ADD CONSTRAINT fk_customer_address FOREIGN KEY (address_id)
REFERENCES address (address_id) ON DELETE RESTRICT ON UPDATE
CASCADE;

ALTER TABLE customer
ADD CONSTRAINT fk_customer_store FOREIGN KEY (store_id)
REFERENCES store (store_id) ON DELETE RESTRICT ON UPDATE
CASCADE;
```

Both of these statements include several ON clauses:

- `ON DELETE RESRICT`, which will cause the server to raise an error if a row is deleted in the parent table (`address` or `store`) which is referenced in the child table (`customer`)
- `ON UPDATE CASCADE`, which will cause the server to propagate a chance to the primary key value of a parent table (`address` or `store`) to the child table (`customer`)

The `ON DELETE RESTRICT` clause protects against orphaned records when rows are deleted from the parent table. To illustrate, let's pick a row in the `address` table and show the data from both the `address` and `customer` tables that share this value:

```
mysql> SELECT c.first_name, c.last_name, c.address_id,
a.address
    -> FROM customer c
    ->   INNER JOIN address a
    ->   ON c.address_id = a.address_id
    -> WHERE a.address_id = 123;
+------------+-----------+------------+---------------------
-------------+
| first_name | last_name | address_id |
address                          |
+------------+-----------+------------+---------------------
-------------+
| SHERRY     | MARSHALL  |        123 | 1987 Coacalco de
Berriozbal Loop |
+------------+-----------+------------+---------------------
-------------+
1 row in set (0.00 sec)
```

The results show that there is a single `customer` row (for Sherry Marshall) whose `address_id` column contains the value 123.

Here's what happens if you try to remove this row from the parent (`address`) table:

```
mysql> DELETE FROM address WHERE address_id = 123;
ERROR 1451 (23000): Cannot delete or update a parent row:
  a foreign key constraint fails (`sakila`.`customer`,
  CONSTRAINT `fk_customer_address` FOREIGN KEY
(`address_id`)
  REFERENCES `address` (`address_id`)
  ON DELETE RESTRICT ON UPDATE CASCADE)
```

Because at least one row in the child table contains the value 123 in the `address_id` column, the `ON DELETE RESTRICT` clause of the foreign key constraint caused the statement to fail.

The `ON UPDATE CASCADE` clause also protects against orphaned records when a primary key value is updated in the parent table, but using a different strategy. Here’s what happens if you modify a value in the `address.address_id` column:

```
mysql> UPDATE address
    -> SET address_id = 9999
    -> WHERE address_id = 123;
Query OK, 1 row affected (0.37 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The statement executed without error, and 1 row was modified. But what happened to Sherry Marshall’s row in the `customer` table? Does it still point to address ID 123, which no longer exists? To find out, let’s run the last query again, but substitute the new value 9999 for the previous value of 123:

```
mysql> SELECT c.first_name, c.last_name, c.address_id,
a.address
    -> FROM customer c
    ->   INNER JOIN address a
    ->   ON c.address_id = a.address_id
    -> WHERE a.address_id = 9999;
+------------+-----------+------------+--------------------
-------------+
| first_name | last_name | address_id |
address                         |
+------------+-----------+------------+--------------------
-------------+
| SHERRY     | MARSHALL  |       9999 | 1987 Coacalco de
Berriozbal Loop |
+------------+-----------+------------+--------------------
-------------+
1 row in set (0.00 sec)
```

As you can see, the same results are returned as before (other than the new address ID value), which means that the value 9999 was automatically updated in the `customer` table. This is known as a *cascade*, and it’s the second mechanism used to protect against orphaned rows.

Along with `RESTRICT` and `CASCADE`, you can also choose `SET NULL`, which will set the foreign key value to NULL in the child table when a row is deleted or updated in the parent table. All together, there are six different options to choose from when defining foreign key constraints:

- `ON DELETE RESTRICT`
- `ON DELETE CASCADE`
- `ON DELETE SET NULL`
- `ON UPDATE RESTRICT`
- `ON UPDATE CASCADE`
- `ON UPDATE SET NULL`

These are optional, so you can choose zero, one, or two (one `ON DELETE` and one `ON UPDATE`) of these when defining your foreign key constraints.

Finally, if you want to remove a primary or foreign key constraint, you can use the `alter table` statement again, except that you specify `drop` instead of `add`. While it is unusual to drop a primary key constraint, foreign key constraints are sometimes dropped during certain maintenance operations and then reestablished.

Test Your Knowledge

Work through the following exercises to test your knowledge of indexes and constraints. When you're done, compare your solutions with those in Appendix C.

Exercise 13-1

Generate an `ALTER TABLE` statement for the `rental` table so that an error will be raised if a row is deleted from the `customer` table having a value found in the `rental.customer_id` column.

Exercise 13-2

Generate a multicolumn index on the `payment` table that could be used by both of the following queries:

```
SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as
datetime);

SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime)
  AND amount < 5;
```

Chapter 14. Views

Well-designed applications generally expose a public interface while keeping implementation details private, thereby enabling future design changes without impacting end users. When designing your database, you can achieve a similar result by keeping your tables private and allowing your users to access data only through a set of *views*. This chapter strives to define what views are, how they are created, and when and how you might want to use them.

What Are Views?

A view is simply a mechanism for querying data. Unlike tables, views do not involve data storage; you won't need to worry about views filling up your disk space. You create a view by assigning a name to a `select` statement, and then storing the query for others to use. Other users can then use your view to access data just as though they were querying tables directly (in fact, they may not even know they are using a view).

As a simple example, let's say that you want to partially obscure the email address in the `customer` table. The marketing department, for example, may need access to email addresses in order to advertise promotions, but otherwise your company's privacy policy dictates that this data be kept secure. Therefore, instead of allowing direct access to the `customer` table, you define a view called `customer_vw` and mandate that all non-marketing personnel use it to access customer data. Here's the view definition:

```
CREATE VIEW customer_vw
 (customer_id,
  first_name,
  last_name,
  email
 )
AS
SELECT
  customer_id,
  first_name,
  last_name,
  concat(substr(email,1,2), '*****', substr(email, -4))
email
FROM customer;
```

The first part of the statement lists the view's column names, which may be different from those of the underlying table. The second part of the statement is a `select` statement, which must contain one expression for each column in the view. The `email` column is generated by taking the first two characters of the email address, concatenated with "*****", and then concatenated with the last 4 characters of the email address.

When the `create view` statement is executed, the database server simply stores the view definition for future use; the query is not executed, and no data is retrieved or stored. Once the view has been created, users can query it just like they would a table, as in:

```
mysql> SELECT first_name, last_name, email
    -> FROM customer_vw;
+-------------+-------------+-------------+
| first_name  | last_name   | email       |
+-------------+-------------+-------------+
| MARY        | SMITH       | MA*****.org |
```

```
| PATRICIA    | JOHNSON      | PA*****.org |
| LINDA       | WILLIAMS     | LI*****.org |
| BARBARA     | JONES        | BA*****.org |
| ELIZABETH   | BROWN        | EL*****.org |
...
| ENRIQUE     | FORSYTHE     | EN*****.org |
| FREDDIE     | DUGGAN       | FR*****.org |
| WADE        | DELVALLE     | WA*****.org |
| AUSTIN      | CINTRON      | AU*****.org |
+-------------+--------------+-------------+
599 rows in set (0.00 sec)
```

Even though the `customer_vw` view definition includes four columns of the `customer` table, the above query retrieves only three of the four. As you'll see later in the chapter, this is an important distinction if some of the columns in your view are attached to functions or subqueries.

From the user's standpoint, a view looks exactly like a table. If you want to know what columns are available in a view, you can use MySQL's (or Oracle's) `describe` command to examine it:

```
mysql> describe customer_vw;
+-------------+----------------------+------+-----+---------
+-------+
| Field       | Type                 | Null | Key | Default
| Extra |
+-------------+----------------------+------+-----+---------
+-------+
| customer_id | smallint(5) unsigned | NO   |     | 0
|       |
| first_name  | varchar(45)          | NO   |     | NULL
|       |
| last_name   | varchar(45)          | NO   |     | NULL
|       |
| email       | varchar(11)          | YES  |     | NULL
|       |
+-------------+----------------------+------+-----+---------
+-------+
4 rows in set (0.00 sec)
```

You are free to use any clauses of the `select` statement when querying through a view, including `group by`, `having`, and `order by`. Here’s an example:

```
mysql> SELECT first_name, count(*), min(last_name),
max(last_name)
    -> FROM customer_vw
    -> WHERE first_name LIKE 'J%'
    -> GROUP BY first_name
    -> HAVING count(*) > 1
    -> ORDER BY 1;
+------------+----------+----------------+----------------+
| first_name | count(*) | min(last_name) | max(last_name) |
+------------+----------+----------------+----------------+
| JAMIE      |        2 | RICE           | WAUGH          |
| JESSIE     |        2 | BANKS          | MILAM          |
+------------+----------+----------------+----------------+
2 rows in set (0.00 sec)
```

In addition, you can join views to other tables (or even to other views) within a query, as in:

```
mysql> SELECT cv.first_name, cv.last_name, p.amount
    -> FROM customer_vw cv
    ->   INNER JOIN payment p
    ->   ON cv.customer_id = p.customer_id
    -> WHERE p.amount >= 11;
+------------+-----------+--------+
| first_name | last_name | amount |
+------------+-----------+--------+
| KAREN      | JACKSON   |  11.99 |
| VICTORIA   | GIBSON    |  11.99 |
| VANESSA    | SIMS      |  11.99 |
| ALMA       | AUSTIN    |  11.99 |
| ROSEMARY   | SCHMIDT   |  11.99 |
| TANYA      | GILBERT   |  11.99 |
| RICHARD    | MCCRARY   |  11.99 |
| NICHOLAS   | BARFIELD  |  11.99 |
| KENT       | ARSENAULT |  11.99 |
| TERRANCE   | ROUSH     |  11.99 |
+------------+-----------+--------+
10 rows in set (0.01 sec)
```

This query joins the `customer_vw` view to the `payment` table in order to find customers who have paid $11 or more for a film rental.

Why Use Views?

In the previous section, I demonstrated a simple view whose sole purpose was to mask the contents of the `customer.email` column. While views are often employed for this purpose, there are many reasons for using views, as detailed in the following subsections.

Data Security

If you create a table and allow users to query it, they will be able to access every column and every row in the table. As I pointed out earlier, however, your table may include some columns that contain sensitive data, such as identification numbers or credit card numbers; not only is it a bad idea to expose such data to all users, but also it might violate your company's privacy policies, or even state or federal laws, to do so.

The best approach for these situations is to keep the table private (i.e., don't grant `select` permission to any users) and then to create one or more views that either omit or obscure (such as the `'*****'` approach taken with the `customer_vw.email` column) the sensitive columns. You may also constrain which *rows* a set of users may access by adding a `where` clause to your view definition. For example, the next view definition excludes inactive customers:

```
CREATE VIEW active_customer_vw
 (customer_id,
  first_name,
  last_name,
  email
 )
AS
SELECT
  customer_id,
  first_name,
  last_name,
  concat(substr(email,1,2), '*****', substr(email, -4))
email
FROM customer
WHERE active = 1;
```

If you provide this view to your marketing department, they will be able to avoid sending information to inactive customers, because the condition in the view's `where` clause will always be included in their queries.

> **NOTE**
>
> Oracle Database users have another option for securing both rows and columns of a table: Virtual Private Database (VPD). VPD allows you to attach policies to your tables, after which the server will modify a user's query as necessary to enforce the policies. For example, if you enact a policy that members of the sales and marketing departments can see only active customers, then the condition `active = 1` will be added to all of their queries against the `customer` table.

Data Aggregation

Reporting applications generally require aggregated data, and views are a great way to make it appear as though data is being pre-aggregated and stored in the database. As an example, let's say that an application generates a report each month showing the total sales for each film category, so that the managers can decide what new films to add to inventory. Rather than allowing the application developers to write queries against the base tables, you could provide them with the following view:

```
CREATE VIEW sales_by_film_category
AS
SELECT
  c.name AS category,
  SUM(p.amount) AS total_sales
FROM payment AS p
  INNER JOIN rental AS r ON p.rental_id = r.rental_id
  INNER JOIN inventory AS i ON r.inventory_id =
i.inventory_id
  INNER JOIN film AS f ON i.film_id = f.film_id
  INNER JOIN film_category AS fc ON f.film_id = fc.film_id
  INNER JOIN category AS c ON fc.category_id = c.category_id
GROUP BY c.name
ORDER BY total_sales DESC;[1]
```

Using this approach gives you a great deal of flexibility as a database designer. If you decide at some point in the future that query performance would improve dramatically if the data were preaggregated in a table rather than summed using a view, you could create a `film_category_sales` table, load it with aggregated data, and modify the `sales_by_film_category` view definition to retrieve data from this table. Afterward, all queries that use the `sales_by_film_category` view will retrieve data from the new `film_category_sales` table, meaning that users will see a performance improvement without needing to modify their queries.

Hiding Complexity

One of the most common reasons for deploying views is to shield end users from complexity. For example, let’s say that a report is created each month showing information about all of the films, along with the film category, the number of actors appearing in the film, the total number of copies in inventory, and the number of rentals for each film. Rather than expecting the report designer to navigate six different tables to gather the necessary data, you could provide a view that looks as follows:

```
CREATE VIEW film_stats
AS
SELECT f.film_id, f.title, f.description, f.rating,
 (SELECT c.name
  FROM category c
    INNER JOIN film_category fc
    ON c.category_id = fc.category_id
  WHERE fc.film_id = f.film_id) category_name,
 (SELECT count(*)
  FROM film_actor fa
  WHERE fa.film_id = f.film_id
 ) num_actors,
 (SELECT count(*)
  FROM inventory i
  WHERE i.film_id = f.film_id
 ) inventory_cnt,
 (SELECT count(*)
  FROM inventory i
    INNER JOIN rental r
    ON i.inventory_id = r.inventory_id
  WHERE i.film_id = f.film_id
 ) num_rentals
FROM film f;
```

This view definition is interesting because even though data from six different tables can be retrieved through the view, the `from` clause of the query only has one table (`film`). Data from the other five tables are generated using scalar subqueries. If someone uses this view but does *not* reference the `category_name`, `num_actors`, `inventory_cnt`, or `num_rentals` column, then none of the subqueries will be executed. This approach allows the view to be used for supplying descriptive information from the `film` table without unnecessarily joining five other tables.

Joining Partitioned Data

Some database designs break large tables into multiple pieces in order to improve performance. For example, if the `payment` table became large, the designers may decide to break it into two tables: `payment_current`, which holds the latest six months' of data, and `payment_historic`, which holds all data up to six months ago. If a customer wants to see all the payments for a particular customer, you would need to query both tables. By creating a view that queries both tables and combines the results together, however, you can make it look like all payment data is stored in a single table. Here's the view definition:

```
CREATE VIEW payment_all
 (payment_id,
  customer_id,
  staff_id,
  rental_id,
  amount,
  payment_date,
  last_update
 )
AS
SELECT payment_id, customer_id, staff_id, rental_id,
  amount, payment_date, last_update
FROM payment_historic
UNION ALL
SELECT payment_id, customer_id, staff_id, rental_id,
  amount, payment_date, last_update
FROM payment_current;
```

Using a view in this case is a good idea because it allows the designers to change the structure of the underlying data without the need to force all database users to modify their queries.

Updatable Views

If you provide users with a set of views to use for data retrieval, what should you do if the users also need to modify the same data? It might seem a bit strange, for example, to force the users to retrieve data using a view, but then allow them to directly modify the underlying table using `update` or `insert` statements. For this purpose, MySQL, Oracle Database, and SQL Server all allow you to modify data through a view, as long as you abide by certain restrictions. In the case of MySQL, a view is updatable if the following conditions are met:

- No aggregate functions are used (`max()`, `min()`, `avg()`, etc.).
- The view does not employ `group by` or `having` clauses.
- No subqueries exist in the `select` or `from` clause, and any subqueries in the `where` clause do not refer to tables in the `from` clause.
- The view does not utilize `union`, `union all`, or `distinct`.
- The `from` clause includes at least one table or updatable view.
- The `from` clause uses only inner joins if there is more than one table or view.

To demonstrate the utility of updatable views, it might be best to start with a simple view definition and then to move to a more complex view.

Updating Simple Views

The view at the beginning of the chapter is about as simple as it gets, so let's start there:

```
CREATE VIEW customer_vw
 (customer_id,
  first_name,
  last_name,
  email
 )
AS
SELECT
  customer_id,
  first_name,
  last_name,
  concat(substr(email,1,2), '*****', substr(email, -4))
email
FROM customer;
```

The `customer_vw` view queries a single table, and only one of the four columns is derived via an expression. This view definition doesn't violate any of the restrictions listed earlier, so you can use it to modify data in the `customer` table. Let's use the view to update Mary Smith's last name to Smith-Allen:

```
mysql> UPDATE customer_vw
    -> SET last_name = 'SMITH-ALLEN'
    -> WHERE customer_id = 1;
Query OK, 1 row affected (0.11 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

As you can see, the statement claims to have modified one row, but let's check the underlying `customer` table just to be sure:

```
mysql> SELECT first_name, last_name, email
    -> FROM customer
    -> WHERE customer_id = 1;
+------------+-------------+-------------------------------+
| first_name | last_name   | email                         |
+------------+-------------+-------------------------------+
| MARY       | SMITH-ALLEN | MARY.SMITH@sakilacustomer.org |
+------------+-------------+-------------------------------+
1 row in set (0.00 sec)
```

While you can modify most of the columns in the view in this fashion, you will not be able to modify the `email` column, since it is derived from an expression:

```
mysql> UPDATE customer_vw
    -> SET email = 'MARY.SMITH-ALLEN@sakilacustomer.org'
    -> WHERE customer_id = 1;
ERROR 1348 (HY000): Column 'email' is not updatable
```

In this case, it may not be a bad thing, since the main reason for creating the view was to obscure the email addresses.

If you want to insert data using the `customer_vw` view, you are out of luck; views that contain derived columns cannot be used for inserting data, even if the derived columns are not included in the statement. For example, the next statement attempts to populate only the `customer_id`, `first_name`, and `last_name` columns using the `customer_vw` view:

```
mysql> INSERT INTO customer_vw
    ->  (customer_id,
    ->   first_name,
    ->   last_name)
    -> VALUES (99999,'ROBERT','SIMPSON');
ERROR 1471 (HY000): The target table customer_vw of the
INSERT
is not insertable-into
```

Now that you have seen the limitations of simple views, the next section will demonstrate the use of a view that joins multiple tables.

Updating Complex Views

While single-table views are certainly common, many of the views that you come across will include multiple tables in the `from` clause of the underlying query. The next view, for example, joins the `customer`, `address`, `city`,and `country` tables so that all the data for customers can be easily queried:

```
CREATE VIEW customer_details
AS
SELECT c.customer_id,
  c.store_id,
  c.first_name,
  c.last_name,
  c.address_id,
  c.active,
  c.create_date,
  a.address,
  ct.city,
  cn.country,
  a.postal_code
FROM customer c
  INNER JOIN address a
  ON c.address_id = a.address_id
  INNER JOIN city ct
  ON a.city_id = ct.city_id
  INNER JOIN country cn
  ON ct.country_id = cn.country_id;
```

You may use this view to update data in either the `customer` or the `address` table, as the following statements demonstrate:

```
mysql> UPDATE customer_details
    -> SET last_name = 'SMITH-ALLEN', active = 0
    -> WHERE customer_id = 1;
Query OK, 1 row affected (0.10 sec)
```

```
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE customer_details
    -> SET address = '999 Mockingbird Lane'
    -> WHERE customer_id = 1;
Query OK, 1 row affected (0.06 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The first statement modifies the `customer.last_name` and `customer.active` columns, whereas the second statement modifies the `address.address` column. You might be wondering what happens if you try to update columns from *both* tables in a single statement, so let's find out:

```
mysql> UPDATE customer_details
    -> SET last_name = 'SMITH-ALLEN',
    ->   active = 0,
    ->   address = '999 Mockingbird Lane'
    -> WHERE customer_id = 1;
ERROR 1393 (HY000): Can not modify more than one base table
  through a join view 'sakila.customer_details'
```

As you can see, you are allowed to modify both of the underlying tables separately, but not within a single statement. Next, let's try to *insert* data into both tables for some new customers (`customer_id = 9998 and 9999`):

```
mysql> INSERT INTO customer_details
    ->  (customer_id, store_id, first_name, last_name,
    ->   address_id, active, create_date)
    -> VALUES (9998, 1, 'BRIAN', 'SALAZAR', 5, 1, now());
Query OK, 1 row affected (0.23 sec)
```

This statement, which only populates columns from the `customer` table, works fine. Let's see what happens if we expand the column list to also include a column from the `address` table:

```
mysql> INSERT INTO customer_details
    ->  (customer_id, store_id, first_name, last_name,
    ->   address_id, active, create_date, address)
    -> VALUES (9999, 2, 'THOMAS', 'BISHOP', 7, 1, now(),
    ->  '999 Mockingbird Lane');
ERROR 1393 (HY000): Can not modify more than one base table
  through a join view 'sakila.customer_details'
```

This version, which includes columns spanning two different tables, raises an exception. In order to insert data through a complex view, you would need to know from where each column is sourced. Since many views are created to hide complexity from end users, this seems to defeat the purpose if the users need to have explicit knowledge of the view definition.

> **NOTE**
>
> Oracle Database and SQL Server also allow data to be inserted and updated through views, but, like MySQL, there are many restrictions. If you are willing to write some PL/SQL or Transact-SQL, however, you can use a feature called *instead-of triggers*, which allows you to essentially intercept `insert`, `update`, and `delete` statements against a view, and write custom code to incorporate the changes. Without this type of feature, there are usually too many restrictions to make updating through views a feasible strategy for nontrivial applications.

Test Your Knowledge

Test your understanding of views by working through the following exercises. When you’re done, compare your solutions with those in Appendix C.

Exercise 14-1

Create a view definition that can be used by the following query to generate the given results:

```
SELECT category_name, title, first_name, last_name
FROM film_ctgry_actor
WHERE last_name = 'FAWCETT';

+---------------------+---------------+------------+--------
---+
| title               | category_name | first_name |
last_name |
+---------------------+---------------+------------+--------
---+
| ACE GOLDFINGER      | Horror        | BOB        |
FAWCETT   |
| ADAPTATION HOLES    | Documentary   | BOB        |
FAWCETT   |
| CHINATOWN GLADIATOR | New           | BOB        |
FAWCETT   |
| CIRCUS YOUTH        | Children      | BOB        |
FAWCETT   |
| CONTROL ANTHEM      | Comedy        | BOB        |
FAWCETT   |
| DARES PLUTO         | Animation     | BOB        |
FAWCETT   |
| DARN FORRESTER      | Action        | BOB        |
FAWCETT   |
| DAZED PUNK          | Games         | BOB        |
FAWCETT   |
| DYNAMITE TARZAN     | Classics      | BOB        |
FAWCETT   |
| HATE HANDICAP       | Comedy        | BOB        |
FAWCETT   |
| HOMICIDE PEACH      | Family        | BOB        |
FAWCETT   |
| JACKET FRISCO       | Drama         | BOB        |
FAWCETT   |
| JUMANJI BLADE       | New           | BOB        |
FAWCETT   |
| LAWLESS VISION      | Animation     | BOB        |
FAWCETT   |
| LEATHERNECKS DWARFS | Travel        | BOB        |
FAWCETT   |
| OSCAR GOLD          | Animation     | BOB        |
FAWCETT   |
| PELICAN COMFORTS    | Documentary   | BOB        |
```

```
FAWCETT    |
| PERSONAL LADYBUGS   | Music         | BOB         |
FAWCETT    |
| RAGING AIRPLANE     | Sci-Fi        | BOB         |
FAWCETT    |
| RUN PACIFIC         | New           | BOB         |
FAWCETT    |
| RUNNER MADIGAN      | Music         | BOB         |
FAWCETT    |
| SADDLE ANTITRUST    | Comedy        | BOB         |
FAWCETT    |
| SCORPION APOLLO     | Drama         | BOB         |
FAWCETT    |
| SHAWSHANK BUBBLE    | Travel        | BOB         |
FAWCETT    |
| TAXI KICK           | Music         | BOB         |
FAWCETT    |
| BERETS AGENT        | Action        | JULIA       |
FAWCETT    |
| BOILED DARES        | Travel        | JULIA       |
FAWCETT    |
| CHISUM BEHAVIOR     | Family        | JULIA       |
FAWCETT    |
| CLOSER BANG         | Comedy        | JULIA       |
FAWCETT    |
| DAY UNFAITHFUL      | New           | JULIA       |
FAWCETT    |
| HOPE TOOTSIE        | Classics      | JULIA       |
FAWCETT    |
| LUKE MUMMY          | Animation     | JULIA       |
FAWCETT    |
| MULAN MOON          | Comedy        | JULIA       |
FAWCETT    |
| OPUS ICE            | Foreign       | JULIA       |
FAWCETT    |
| POLLOCK DELIVERANCE | Foreign       | JULIA       |
FAWCETT    |
| RIDGEMONT SUBMARINE | New           | JULIA       |
FAWCETT    |
| SHANGHAI TYCOON     | Travel        | JULIA       |
FAWCETT    |
| SHAWSHANK BUBBLE    | Travel        | JULIA       |
FAWCETT    |
| THEORY MERMAID      | Animation     | JULIA       |
FAWCETT    |
| WAIT CIDER          | Animation     | JULIA       |
FAWCETT    |
+---------------------+---------------+-------------+--------
```

```
---+
40 rows in set (0.00 sec)
```

Exercise 14-2

The film rental company manager would like to have a report that includes the name of every country, along with the total payments for all customers who live in each country. Generate a view definition that queries the `country` table and uses a scalar subquery to calculate a value for a column named `tot_payments`

---

1 This view definition is included in the Sakila sample database, along with six others, several of which will be used in upcoming examples.

Chapter 15. Metadata

Along with storing all of the data that various users insert into a database, a database server also needs to store information about all of the database objects (tables, views, indexes, etc.) that were created to store this data. The database server stores this information, not surprisingly, in a database. This chapter discusses how and where this information, known as *metadata*, is stored, how you can access it, and how you can use it to build flexible systems.

Data About Data

Metadata is essentially data about data. Every time you create a database object, the database server needs to record various pieces of information. For example, if you were to create a table with multiple columns, a primary key constraint, three indexes, and a foreign key constraint, the database server would need to store all the following information:

- Table name
- Table storage information (tablespace, initial size, etc.)
- Storage engine
- Column names
- Column data types
- Default column values
- `NOT NULL` column constraints
- Primary key columns
- Primary key name

- Name of primary key index
- Index names
- Index types (B-tree, bitmap)
- Indexed columns
- Index column sort order (ascending or descending)
- Index storage information
- Foreign key name
- Foreign key columns
- Associated table/columns for foreign keys

This data is collectively known as the *data dictionary* or *system catalog*. The database server needs to store this data persistently, and it needs to be able to quickly retrieve this data in order to verify and execute SQL statements. Additionally, the database server must safeguard this data so that it can be modified only via an appropriate mechanism, such as the `alter table` statement.

While standards exist for the exchange of metadata between different servers, every database server uses a different mechanism to publish metadata, such as:

- A set of views, such as Oracle Database's `user_tables` and `all_constraints` views
- A set of system-stored procedures, such as SQL Server's `sp_tables` procedure or Oracle Database's `dbms_metadata` package
- A special database, such as MySQL's `information_schema` database

Along with SQL Server's system-stored procedures, which are a vestige of its Sybase lineage, SQL Server also includes a special schema called `information_schema` that is provided automatically within each database. Both MySQL and SQL Server provide this interface to conform with the ANSI SQL:2003 standard. The remainder of this chapter discusses the `information_schema` objects that are available in MySQL and SQL Server.

Information_Schema

All of the objects available within the `information_schema` database (or schema, in the case of SQL Server) are views. Unlike the `describe` utility, which I used in several chapters of this book as a way to show the structure of various tables and views, the views within `information_schema` can be queried, and, thus, used programmatically (more on this later in the chapter). Here's an example that demonstrates how to retrieve the names of all of the tables in the `sakila` database:

```
mysql> SELECT table_name, table_type
    -> FROM information_schema.tables
    -> WHERE table_schema = 'sakila'
    -> ORDER BY 1;
+----------------------------+------------+
| TABLE_NAME                 | TABLE_TYPE |
+----------------------------+------------+
| actor                      | BASE TABLE |
| actor_info                 | VIEW       |
| address                    | BASE TABLE |
| category                   | BASE TABLE |
| city                       | BASE TABLE |
| country                    | BASE TABLE |
| customer                   | BASE TABLE |
| customer_list              | VIEW       |
| film                       | BASE TABLE |
| film_actor                 | BASE TABLE |
| film_category              | BASE TABLE |
| film_list                  | VIEW       |
| film_text                  | BASE TABLE |
| inventory                  | BASE TABLE |
| language                   | BASE TABLE |
| nicer_but_slower_film_list | VIEW       |
| payment                    | BASE TABLE |
| rental                     | BASE TABLE |
| sales_by_film_category     | VIEW       |
```

```
| sales_by_store             | VIEW       |
| staff                      | BASE TABLE |
| staff_list                 | VIEW       |
| store                      | BASE TABLE |
+----------------------------+------------+
23 rows in set (0.00 sec)
```

As you can see, the `information_schema.tables` view includes both tables and views; if you want to exclude the views, simply add another condition to the `where` clause:

```
mysql> SELECT table_name, table_type
    -> FROM information_schema.tables
    -> WHERE table_schema = 'sakila'
    ->   AND table_type = 'BASE TABLE'
    -> ORDER BY 1;
+---------------+------------+
| TABLE_NAME    | TABLE_TYPE |
+---------------+------------+
| actor         | BASE TABLE |
| address       | BASE TABLE |
| category      | BASE TABLE |
| city          | BASE TABLE |
| country       | BASE TABLE |
| customer      | BASE TABLE |
| film          | BASE TABLE |
| film_actor    | BASE TABLE |
| film_category | BASE TABLE |
| film_text     | BASE TABLE |
| inventory     | BASE TABLE |
| language      | BASE TABLE |
| payment       | BASE TABLE |
| rental        | BASE TABLE |
| staff         | BASE TABLE |
| store         | BASE TABLE |
+---------------+------------+
16 rows in set (0.00 sec)
```

If you are only interested in information about views, you can query `information_schema.views`. Along with the view names, you can retrieve additional information, such as a flag that shows whether a view is updatable:

```
mysql> SELECT table_name, is_updatable
    -> FROM information_schema.views
    -> WHERE table_schema = 'sakila'
```

```
    -> ORDER BY 1;
+----------------------------+--------------+
| TABLE_NAME                 | IS_UPDATABLE |
+----------------------------+--------------+
| actor_info                 | NO           |
| customer_list              | YES          |
| film_list                  | NO           |
| nicer_but_slower_film_list | NO           |
| sales_by_film_category     | NO           |
| sales_by_store             | NO           |
| staff_list                 | YES          |
+----------------------------+--------------+
7 rows in set (0.00 sec)
```

Column information for both tables and views is available via the `columns` view. The following query shows column information for the `film` table:

```
mysql> SELECT column_name, data_type,
    ->   character_maximum_length char_max_len,
    ->   numeric_precision num_prcsn, numeric_scale num_scale
    -> FROM information_schema.columns
    -> WHERE table_schema = 'sakila' AND table_name = 'film'
    -> ORDER BY ordinal_position;
+----------------------+-----------+--------------+-----------+----
-------+
| COLUMN_NAME          | DATA_TYPE | char_max_len | num_prcsn |
num_scale |
+----------------------+-----------+--------------+-----------+----
-------+
| film_id              | smallint  |         NULL |         5
|         0 |
| title                | varchar   |          255 |      NULL
|      NULL |
| description          | text      |        65535 |      NULL
|      NULL |
| release_year         | year      |         NULL |      NULL
|      NULL |
| language_id          | tinyint   |         NULL |         3
|         0 |
| original_language_id | tinyint   |         NULL |         3
|         0 |
| rental_duration      | tinyint   |         NULL |         3
|         0 |
| rental_rate          | decimal   |         NULL |         4
|         2 |
| length               | smallint  |         NULL |         5
|         0 |
| replacement_cost     | decimal   |         NULL |         5
|         2 |
| rating               | enum      |            5 |      NULL
```

```
|      NULL |
| special_features     | set       |           54 |      NULL
|      NULL |
| last_update          | timestamp |         NULL |      NULL
|      NULL |
+----------------------+-----------+-------------+----------+----
-------+
13 rows in set (0.00 sec)
```

The `ordinal_position` column is included merely as a means to retrieve the columns in the order in which they were added to the table.

You can retrieve information about a table's indexes via the `information_schema.statistics` view as demonstrated by the following query, which retrieves information for the indexes built on the `rental` table:

```
mysql> SELECT index_name, non_unique, seq_in_index, column_name
    -> FROM information_schema.statistics
    -> WHERE table_schema = 'sakila' AND table_name = 'rental'
    -> ORDER BY 1, 3;
+---------------------+------------+--------------+--------------+
| INDEX_NAME          | NON_UNIQUE | SEQ_IN_INDEX | COLUMN_NAME  |
+---------------------+------------+--------------+--------------+
| idx_fk_customer_id  |          1 |            1 | customer_id  |
| idx_fk_inventory_id |          1 |            1 | inventory_id |
| idx_fk_staff_id     |          1 |            1 | staff_id     |
| PRIMARY             |          0 |            1 | rental_id    |
| rental_date         |          0 |            1 | rental_date  |
| rental_date         |          0 |            2 | inventory_id |
| rental_date         |          0 |            3 | customer_id  |
+---------------------+------------+--------------+--------------+
7 rows in set (0.02 sec)
```

The `rental` table has a total of five indexes, one of which has three columns (`rental_date`) and one of which is a unique index (`PRIMARY`) used for the primary key constraint.

You can retrieve the different types of constraints (foreign key, primary key, unique) that have been created via the `information_schema.table_constraints` view. Here's a query that retrieves all of the constraints in the `sakila` schema:

```
mysql> SELECT constraint_name, table_name, constraint_type
    -> FROM information_schema.table_constraints
    -> WHERE table_schema = 'sakila'
    -> ORDER BY 3,1;
+--------------------------+--------------+-----------------+
| constraint_name          | table_name   | constraint_type |
+--------------------------+--------------+-----------------+
| fk_address_city          | address      | FOREIGN KEY     |
| fk_city_country          | city         | FOREIGN KEY     |
| fk_customer_address      | customer     | FOREIGN KEY     |
| fk_customer_store        | customer     | FOREIGN KEY     |
| fk_film_actor_actor      | film_actor   | FOREIGN KEY     |
| fk_film_actor_film       | film_actor   | FOREIGN KEY     |
| fk_film_category_category | film_category | FOREIGN KEY   |
| fk_film_category_film    | film_category | FOREIGN KEY    |
| fk_film_language         | film         | FOREIGN KEY     |
| fk_film_language_original | film        | FOREIGN KEY     |
| fk_inventory_film        | inventory    | FOREIGN KEY     |
| fk_inventory_store       | inventory    | FOREIGN KEY     |
| fk_payment_customer      | payment      | FOREIGN KEY     |
| fk_payment_rental        | payment      | FOREIGN KEY     |
| fk_payment_staff         | payment      | FOREIGN KEY     |
| fk_rental_customer       | rental       | FOREIGN KEY     |
| fk_rental_inventory      | rental       | FOREIGN KEY     |
| fk_rental_staff          | rental       | FOREIGN KEY     |
| fk_staff_address         | staff        | FOREIGN KEY     |
| fk_staff_store           | staff        | FOREIGN KEY     |
| fk_store_address         | store        | FOREIGN KEY     |
| fk_store_staff           | store        | FOREIGN KEY     |
| PRIMARY                  | film         | PRIMARY KEY     |
| PRIMARY                  | film_actor   | PRIMARY KEY     |
| PRIMARY                  | staff        | PRIMARY KEY     |
| PRIMARY                  | film_category | PRIMARY KEY    |
| PRIMARY                  | store        | PRIMARY KEY     |
| PRIMARY                  | actor        | PRIMARY KEY     |
| PRIMARY                  | film_text    | PRIMARY KEY     |
| PRIMARY                  | address      | PRIMARY KEY     |
| PRIMARY                  | inventory    | PRIMARY KEY     |
| PRIMARY                  | customer     | PRIMARY KEY     |
| PRIMARY                  | category     | PRIMARY KEY     |
| PRIMARY                  | language     | PRIMARY KEY     |
| PRIMARY                  | city         | PRIMARY KEY     |
| PRIMARY                  | payment      | PRIMARY KEY     |
| PRIMARY                  | country      | PRIMARY KEY     |
| PRIMARY                  | rental       | PRIMARY KEY     |
| idx_email                | customer     | UNIQUE          |
| idx_unique_manager       | store        | UNIQUE          |
| rental_date              | rental       | UNIQUE          |
```

```
+---------------------------+---------------+-----------------+
41 rows in set (0.02 sec)
```

Table 15-1 shows many of the `information_schema` views that are available in MySQL version 8.0.

Table 15-1. Information_schema views

View name	Provides information about…
`Schemata`	Databases
`Tables`	Tables and views
`Columns`	Columns of tables and views
`Statistics`	Indexes
`User_Privileges`	Who has privileges on which schema objects
`Schema_Privileges`	Who has privileges on which databases
`Table_Privileges`	Who has privileges on which tables

View name	Provides information about…
`Column_Privileges`	Who has privileges on which columns of which tables
`Character_Sets`	What character sets are available
`Collations`	What collations are available for which character sets
`Collation_Character_Set_Applicability`	Which character sets are available for which collation
`Table_Constraints`	The unique, foreign key, and primary key constraints
`Key_Column_Usage`	The constraints associated with each key column
`Routines`	Stored routines (procedures and functions)

View name	Provides information about…
`Views`	Views
`Triggers`	Table triggers
`Plugins`	Server plug-ins
`Engines`	Available storage engines
`Partitions`	Table partitions
`Events`	Scheduled events
`ProcessList`	Running processes
`Referential_Constraints`	Foreign keys

View name	Provides information about…
`Parameters`	Stored procedure and function parameters
`Profiling`	User profiling information

While some of these views, such as `engines`, `events`, and `plugins`, are specific to MySQL, many of these views are available in SQL Server as well. If you are using Oracle Database, please consult the online Oracle Database Reference Guide for information about the `user_`, `all_`, and `dba_` views.

Working with Metadata

As I mentioned earlier, having the ability to retrieve information about your schema objects via SQL queries opens up some interesting possibilities. This section shows several ways in which you can make use of metadata in your applications.

Schema Generation Scripts

While some project teams include a full-time database designer who oversees the design and implementation of the database, many projects take the "design-by-committee" approach, allowing multiple people to create database objects. After several weeks or months of development, you may need to generate a script that will create the various tables, indexes, views, and so on that the team has deployed. Although a variety of tools and utilities will generate these types of scripts for you, you can also query the `information_schema` views and generate the script yourself.

As an example, let’s build a script that will create the `sakila.category` table. Here’s the command used to build the table, which I extracted from the script used to build the example database:

```
CREATE TABLE category (
  category_id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(25) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (category_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Although it would certainly be easier to generate the script with the use of a procedural language (e.g., Transact-SQL or Java), since this is a book about SQL I’m going to write a single query that will generate the `create table` statement. The first step is to query the `information_schema.columns` table to retrieve information about the columns in the table:

```
mysql> SELECT 'CREATE TABLE category (' create_table_statement
    -> UNION ALL
    -> SELECT cols.txt
    -> FROM
    ->  (SELECT concat('  ',column_name, ' ', column_type,
    ->    CASE
    ->      WHEN is_nullable = 'NO' THEN ' not null'
    ->      ELSE ''
    ->    END,
    ->    CASE
    ->      WHEN extra IS NOT NULL AND extra LIKE
'DEFAULT_GENERATED%'
    ->       THEN concat(' DEFAULT
',column_default,substr(extra,18))
    ->      WHEN extra IS NOT NULL THEN concat(' ', extra)
    ->      ELSE ''
    ->    END,
    ->    ',') txt
    ->   FROM information_schema.columns
    ->   WHERE table_schema = 'sakila' AND table_name = 'category'
    ->   ORDER BY ordinal_position
    ->  ) cols
    -> UNION ALL
    -> SELECT ')';
+-------------------------------------------------------------------
-----+
```

```
|
create_table_statement                                                |

+-------------------------------------------------------------------
-----+
| CREATE TABLE category
(                                                |
|   category_id tinyint(3) unsigned not null
auto_increment,            |
|   name varchar(25) not null
,                                          |
|   last_update timestamp not null DEFAULT CURRENT_TIMESTAMP
|
|     on update CURRENT_TIMESTAMP,
|
|
)
 |
+-------------------------------------------------------------------
-----+
5 rows in set (0.00 sec)
```

Well, that got us pretty close; we just need to add queries against the `table_constraints` and `key_column_usage` views to retrieve information about the primary key constraint:

```
mysql> SELECT 'CREATE TABLE category (' create_table_statement
    -> UNION ALL
    -> SELECT cols.txt
    -> FROM
    ->  (SELECT concat('  ',column_name, ' ', column_type,
    ->    CASE
    ->      WHEN is_nullable = 'NO' THEN ' not null'
    ->      ELSE ''
    ->    END,
    ->    CASE
    ->      WHEN extra IS NOT NULL AND extra LIKE
'DEFAULT_GENERATED%'
    ->        THEN concat(' DEFAULT
',column_default,substr(extra,18))
    ->      WHEN extra IS NOT NULL THEN concat(' ', extra)
    ->      ELSE ''
    ->    END,
    ->    ',') txt
    ->   FROM information_schema.columns
    ->   WHERE table_schema = 'sakila' AND table_name = 'category'
    ->   ORDER BY ordinal_position
    ->  ) cols
    -> UNION ALL
    -> SELECT concat('  constraint primary key (')
```

```
    -> FROM information_schema.table_constraints
    -> WHERE table_schema = 'sakila' AND table_name = 'category'
    ->   AND constraint_type = 'PRIMARY KEY'
    -> UNION ALL
    -> SELECT cols.txt
    -> FROM
    ->  (SELECT concat(CASE WHEN ordinal_position > 1 THEN '   ,'
    ->     ELSE '    ' END, column_name) txt
    ->   FROM information_schema.key_column_usage
    ->   WHERE table_schema = 'sakila' AND table_name = 'category'
    ->     AND constraint_name = 'PRIMARY'
    ->   ORDER BY ordinal_position
    ->  ) cols
    -> UNION ALL
    -> SELECT '  )'
    -> UNION ALL
    -> SELECT ')';
+------------------------------------------------------------------
-----+
|
create_table_statement
 |
+------------------------------------------------------------------
-----+
| CREATE TABLE category
(                                                |
|   category_id tinyint(3) unsigned not null
auto_increment,            |
|   name varchar(25) not null
,                                          |
|   last_update timestamp not null DEFAULT CURRENT_TIMESTAMP
|
|     on update CURRENT_TIMESTAMP,
|
|   constraint primary key (
      |
|
category_id                                                      |
|
)
|
|
)
 |
+------------------------------------------------------------------
-----+
8 rows in set (0.02 sec)
```

To see whether the statement is properly formed, I’ll paste the query output into the `mysql` tool (I’ve changed the table name to `category2` so that it won’t step on our existing table):

```
mysql> CREATE TABLE category2 (
    ->   category_id tinyint(3) unsigned not null auto_increment,
    ->   name varchar(25) not null ,
    ->   last_update timestamp not null DEFAULT CURRENT_TIMESTAMP
    ->     on update CURRENT_TIMESTAMP,
    ->   constraint primary key (
    ->     category_id
    ->   )
    -> );
Query OK, 0 rows affected (0.61 sec)
```

The statement executed without errors, and there is now a `category2` table in the `sakila` database. In order for the query to generate a well-formed `create table` statement for *any* table, more work is required (such as handling indexes and foreign key constraints), but I’ll leave that as an exercise.

Deployment Verification

Many organizations allow for database maintenance windows, wherein existing database objects may be administered (such as adding/dropping partitions) and new schema objects and code can be deployed. After the deployment scripts have been run, it’s a good idea to run a verification script to ensure that the new schema objects are in place with the appropriate columns, indexes, primary keys, and so forth. Here’s a query that returns the number of columns, number of indexes, and number of primary key constraints (`0` or `1`) for each table in the `sakila` schema:

```
mysql> SELECT tbl.table_name,
    ->  (SELECT count(*) FROM information_schema.columns clm
    ->   WHERE clm.table_schema = tbl.table_schema
    ->     AND clm.table_name = tbl.table_name) num_columns,
    ->  (SELECT count(*) FROM information_schema.statistics sta
    ->   WHERE sta.table_schema = tbl.table_schema
    ->     AND sta.table_name = tbl.table_name) num_indexes,
    ->  (SELECT count(*) FROM information_schema.table_constraints
tc
    ->   WHERE tc.table_schema = tbl.table_schema
    ->     AND tc.table_name = tbl.table_name
```

```
    ->     AND tc.constraint_type = 'PRIMARY KEY') num_primary_keys
    -> FROM information_schema.tables tbl
    -> WHERE tbl.table_schema = 'sakila' AND tbl.table_type = 'BASE
TABLE'
    -> ORDER BY 1;
+---------------+-------------+-------------+------------------+
| TABLE_NAME    | num_columns | num_indexes | num_primary_keys |
+---------------+-------------+-------------+------------------+
| actor         |           4 |           2 |                1 |
| address       |           9 |           3 |                1 |
| category      |           3 |           1 |                1 |
| city          |           4 |           2 |                1 |
| country       |           3 |           1 |                1 |
| customer      |           9 |           7 |                1 |
| film          |          13 |           4 |                1 |
| film_actor    |           3 |           3 |                1 |
| film_category |           3 |           3 |                1 |
| film_text     |           3 |           3 |                1 |
| inventory     |           4 |           4 |                1 |
| language      |           3 |           1 |                1 |
| payment       |           7 |           4 |                1 |
| rental        |           7 |           7 |                1 |
| staff         |          11 |           3 |                1 |
| store         |           4 |           3 |                1 |
+---------------+-------------+-------------+------------------+
16 rows in set (0.01 sec)
```

You could execute this statement before and after the deployment and then verify any differences between the two sets of results before declaring the deployment a success.

Dynamic SQL Generation

Some languages, such as Oracle’s PL/SQL and Microsoft’s Transact-SQL, are supersets of the SQL language, meaning that they include SQL statements in their grammar along with the usual procedural constructs, such as “if-then-else” and “while.” Other languages, such as Java, include the ability to interface with a relational database, but do not include SQL statements in the grammar, meaning that all SQL statements must be contained within strings.

Therefore, most relational database servers, including SQL Server, Oracle Database, and MySQL, allow SQL statements to be submitted to the server as strings. Submitting strings to a database engine rather than utilizing its SQL interface is generally known as *dynamic SQL execution*. Oracle's PL/SQL language, for example, includes an `execute immediate` command, which you can use to submit a string for execution, while SQL Server includes a system stored procedure called `sp_executesql` for executing SQL statements dynamically.

MySQL provides the statements `prepare`, `execute`, and `deallocate` to allow for dynamic SQL execution. Here's a simple example:

```
mysql> SET @qry = 'SELECT customer_id, first_name, last_name FROM
customer';
Query OK, 0 rows affected (0.00 sec)

mysql> PREPARE dynsql1 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> EXECUTE dynsql1;
+-------------+------------+-------------+
| customer_id | first_name | last_name   |
+-------------+------------+-------------+
|         505 | RAFAEL     | ABNEY       |
|         504 | NATHANIEL  | ADAM        |
|          36 | KATHLEEN   | ADAMS       |
|          96 | DIANA      | ALEXANDER   |
...
|          31 | BRENDA     | WRIGHT      |
|         318 | BRIAN      | WYMAN       |
|         402 | LUIS       | YANEZ       |
|         413 | MARVIN     | YEE         |
|          28 | CYNTHIA    | YOUNG       |
+-------------+------------+-------------+
599 rows in set (0.02 sec)

mysql> DEALLOCATE PREPARE dynsql1;
Query OK, 0 rows affected (0.00 sec)
```

The `set` statement simply assigns a string to the `qry` variable, which is then submitted to the database engine (for parsing, security checking, and optimization) using the `prepare` statement. After executing the statement by calling `execute`, the statement must be closed using `deallocate` `prepare`, which frees any database resources (e.g., cursors) that have been utilized during execution.

The next example shows how you could execute a query that includes placeholders so that conditions can be specified at runtime:

```
mysql> SET @qry = 'SELECT customer_id, first_name, last_name
  FROM customer WHERE customer_id = ?';
Query OK, 0 rows affected (0.00 sec)

mysql> PREPARE dynsql2 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> SET @custid = 9;
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql2 USING @custid;
+-------------+------------+-----------+
| customer_id | first_name | last_name |
+-------------+------------+-----------+
|           9 | MARGARET   | MOORE     |
+-------------+------------+-----------+
1 row in set (0.00 sec)

mysql> SET @custid = 145;
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql2 USING @custid;
+-------------+------------+-----------+
| customer_id | first_name | last_name |
+-------------+------------+-----------+
|         145 | LUCILLE    | HOLMES    |
+-------------+------------+-----------+
1 row in set (0.00 sec)

mysql> DEALLOCATE PREPARE dynsql2;
Query OK, 0 rows affected (0.00 sec)
```

In this sequence, the query contains a placeholder (the `?` at the end of the statement) so that the customer ID value can be submitted at runtime. The statement is prepared once and then executed twice, once for customer ID 9, and again for customer ID 145, after which the statement is closed.

What, you may wonder, does this have to do with metadata? Well, if you are going to use dynamic SQL to query a table, why not build the query string using metadata rather than hardcoding the table definition? The following example generates the same dynamic SQL string as the previous example, but it retrieves the column names from the `information_schema.columns` view:

```
mysql> SELECT concat('SELECT ',
    ->   concat_ws(',', cols.col1, cols.col2, cols.col3, cols.col4,
    ->     cols.col5, cols.col6, cols.col7, cols.col8, cols.col9),
    ->   ' FROM customer WHERE customer_id = ?')
    -> INTO @qry
    -> FROM
    ->  (SELECT
    ->     max(CASE WHEN ordinal_position = 1 THEN column_name
    ->       ELSE NULL END) col1,
    ->     max(CASE WHEN ordinal_position = 2 THEN column_name
    ->       ELSE NULL END) col2,
    ->     max(CASE WHEN ordinal_position = 3 THEN column_name
    ->       ELSE NULL END) col3,
    ->     max(CASE WHEN ordinal_position = 4 THEN column_name
    ->       ELSE NULL END) col4,
    ->     max(CASE WHEN ordinal_position = 5 THEN column_name
    ->       ELSE NULL END) col5,
    ->     max(CASE WHEN ordinal_position = 6 THEN column_name
    ->       ELSE NULL END) col6,
    ->     max(CASE WHEN ordinal_position = 7 THEN column_name
    ->       ELSE NULL END) col7,
    ->     max(CASE WHEN ordinal_position = 8 THEN column_name
    ->       ELSE NULL END) col8,
    ->     max(CASE WHEN ordinal_position = 9 THEN column_name
    ->       ELSE NULL END) col9
    ->   FROM information_schema.columns
    ->   WHERE table_schema = 'sakila' AND table_name = 'customer'
    ->   GROUP BY table_name
    ->  ) cols;
Query OK, 1 row affected (0.00 sec)
mysql> SELECT @qry;
mysql> SELECT @qry;
+-------------------------------------------------------------------
--+
| @qry
       |
+-------------------------------------------------------------------
--+
| SELECT customer_id,store_id,first_name,last_name,email,
    address_id,active,create_date,last_update
  FROM customer WHERE customer_id = ?
```

```
|
+------------------------------------------------------------------
--+
1 row in set (0.00 sec)

mysql> PREPARE dynsql3 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> SET @custid = 45;
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql3 USING @custid;
+-------------+----------+------------+-----------+---------------
------------------+------------+--------+---------------------+---
------------------+
| customer_id | store_id | first_name | last_name |
email                            | address_id | active |
create_date         | last_update         |
+-------------+----------+------------+-----------+---------------
------------------+------------+--------+---------------------+---
------------------+
|          45 |        1 | JANET      | PHILLIPS  |
JANET.PHILLIPS@sakilacustomer.org |         49 |      1 | 2006-02-
14 22:04:36 | 2006-02-15 04:57:20 |
+-------------+----------+------------+-----------+---------------
------------------+------------+--------+---------------------+---
------------------+
1 row in set (0.00 sec)

mysql> DEALLOCATE PREPARE dynsql3;
Query OK, 0 rows affected (0.00 sec)
```

The query pivots the first nine columns in the `customer` table, builds a query string using the `concat` and `concat_ws` functions, and assigns the string to the `qry` variable. The query string is then executed as before.

> **NOTE**
>
> Generally, it would be better to generate the query using a procedural language that includes looping constructs, such as Java, PL/SQL, Transact-SQL, or MySQL’s Stored Procedure Language. However, I wanted to demonstrate a pure SQL example, so I had to limit the number of columns retrieved to some reasonable number, which in this example is nine.

Test Your Knowledge

The following exercises are designed to test your understanding of metadata. When you’re finished, please see Appendix C for the solutions.

Exercise 15-1

Write a query that lists all of the indexes in the `sakila` schema. Include the table names.

Exercise 15-2

Write a query that generates output that can be used to create all of the indexes on the `sakila.customer` table. Output should be of the form:

```
"ALTER TABLE <table_name> ADD INDEX <index_name> (<column_list>)"
```

Chapter 16. Analytic Functions

Data volumes have been growing at a staggering pace, and organizations are having difficulty storing all of it, not to mention trying to make sense of it. While data analysis has traditionally occurred outside of the database server, using specialized tools or languages such as Excel, R, and Python, the SQL language includes a robust set of functions useful for analytical processing. Common functionality such as generating rankings or calculating sub-totals can be easily accomplished within the database server without the need to export the data and load it into another tool.

Analytic Function Concepts

After the database server has completed all of the steps necessary to evaluate a query, including joining, filtering, grouping, and sorting, the result set is complete and ready to be returned to the caller. Imagine if you could pause the query execution at this point and take a walk through the result set while it is still held in memory; what types of analysis might you want to do? If your result set contains sales data, perhaps you might want to generate rankings for salespeople or regions, or calculate percentage differences between one time period to another. If you are generating results for a financial report, perhaps you would like to calculate subtotals for each report section, and a grand total for the final section. Using analytic functions, you can do all of these things and more. Before diving into the details, the following subsections describe the mechanisms used by several of the most commonly used analytic functions.

Data Windows

Let's say you have written a query which generates monthly sales totals across an entire year. You may want to find the maximum monthly sales across the full year, which would involve finding the maximum value across the entire result set. However, you may also want to find the maximum monthly sales for each quarter, which would require the result set to be split into four pieces. To accommodate this type of analysis, analytic functions include the ability to group rows into `windows`, which effectively partition the data for use by the analytic function without changing the overall result set. Windows are defined using the `over` clause combined with the `partition by` sub clause, as demonstrated by the following query:

```
mysql> SELECT quarter(payment_date) quarter,
    ->   monthname(payment_date) month_nm,
    ->   sum(amount) monthly_sales,
    ->   max(sum(amount))
    ->     over () max_mnth_sales,
    ->   max(sum(amount))
    ->     over (partition by quarter(payment_date))
max_qrtr_sales
    -> FROM payment
    -> WHERE year(payment_date) = 2005
    -> GROUP BY quarter(payment_date),
monthname(payment_date);
+---------+---------+--------------+---------------+-----
```

```
-----------+
| quarter | month_nm | monthly_sales | max_mnth_sales |
max_qrtr_sales |
+---------+---------+-------------+--------------+-----
-----------+
|       2 | May      |       4824.43 |       28373.89
|        9631.88 |
|       2 | June     |       9631.88 |       28373.89
|        9631.88 |
|       3 | July     |      28373.89 |       28373.89
|       28373.89 |
|       3 | August   |      24072.13 |       28373.89
|       28373.89 |
+---------+---------+-------------+--------------+-----
-----------+
4 rows in set (0.00 sec)
```

This query calculates the monthly sales for each month (`monthly_sales`), and then includes two analytic functions to calculate the maximum monthly sales (`max_mnth_sales`) and the maximum sales within each quarter (`max_qrtr_sales`). Both analytic functions include an `over` clause, but the first one is empty, indicating that the window should include the entire result set, whereas the second one specifies that the window should only include rows within the same quarter. Data windows may contain anywhere from a single row to all of the rows in the result set, and different analytic functions can define different data windows.

Localized Sorting

Along with partitioning your result set into data windows for your analytic functions, you may also specify a sort order. For example, if you want to define a ranking number for each month, where the value 1 is given to the month having the highest sales, you will need to specify which column (or columns) to use for the ranking:

```
mysql> SELECT quarter(payment_date) quarter,
    ->   monthname(payment_date) month_nm,
    ->   sum(amount) monthly_sales,
    ->   rank() over (order by sum(amount) desc) sales_rank
    -> FROM payment
    -> WHERE year(payment_date) = 2005
    -> GROUP BY quarter(payment_date),
monthname(payment_date)
    -> ORDER BY 1, month(payment_date);
+---------+----------+---------------+------------+
| quarter | month_nm | monthly_sales | sales_rank |
+---------+----------+---------------+------------+
|       2 | May      |       4824.43 |          4 |
|       2 | June     |       9631.88 |          3 |
|       3 | July     |      28373.89 |          1 |
|       3 | August   |      24072.13 |          2 |
+---------+----------+---------------+------------+
4 rows in set (0.00 sec)
```

This query includes a call to the `rank` function, which will be covered in the next section, and specifies that the sum of the `amount` column be used to generate the rankings, with the values sorted in descending order. Thus, the month having the highest sales (July, in this case) will be given a ranking of 1.

MULTIPLE ORDER BY CLAUSES

The previous example contains two `order by` clauses; one at the end of the query to determine how the result set should be sorted, and another within the `rank` function to determine how the rankings should be allocated. While it is unfortunate that the same clause is used for different purposes, keep in mind that even if you are using analytic functions with one or more `order by` clauses, you will still need an `order by` clause at the end of your query if you want the result set to be sorted in a particular way.

In some cases, you will want to use both the `partition by` and `order by` sub clauses in the same analytic function call. For example, the previous example can be modified to provide a different set of rankings per quarter, rather than a single ranking across the entire result set:

```
mysql> SELECT quarter(payment_date) quarter,
    ->   monthname(payment_date) month_nm,
    ->   sum(amount) monthly_sales,
    ->   rank() over (partition by quarter(payment_date)
    ->     order by sum(amount) desc) qtr_sales_rank
    -> FROM payment
    -> WHERE year(payment_date) = 2005
    -> GROUP BY quarter(payment_date),
monthname(payment_date)
    -> ORDER BY 1, month(payment_date);
+---------+----------+---------------+----------------+
| quarter | month_nm | monthly_sales | qtr_sales_rank |
+---------+----------+---------------+----------------+
|       2 | May      |       4824.43 |              2 |
|       2 | June     |       9631.88 |              1 |
|       3 | July     |      28373.89 |              1 |
|       3 | August   |      24072.13 |              2 |
+---------+----------+---------------+----------------+
4 rows in set (0.00 sec)
```

While these examples were designed to illustrate the use of the `over` clause, the following sections will describe in detail the various analytic functions.

Ranking

People love to rank things. If you visit your favorite news/sports / travel sites, you’ll see headlines similar to the following:

- Top 10 Vacation Values
- Best Mutual Fund Returns
- Pre-Season College Football Rankings

- Top 100 Songs of All Time

Companies also like to generate rankings, but for more practical purposes. Knowing which products are the best/worst sellers, or which geographic regions generate the least/most revenue help organizations to make strategic decisions.

Ranking Functions

There are multiple ranking functions available in the SQL standard, with each one taking a different approach to how ties are handled:

Row_Number	Returns a unique number for each row, with rankings arbitrarily assigned in case of a tie.
Rank	Returns the same ranking in case of a tie, with gaps in the rankings.
Dense_Rank	Returns the same ranking in case of a tie, with no gaps in the rankings.

Let's look at an example to help illustrate the differences. Let's say that the marketing department wants to identify the top 10 customers so they can be offered a free film rental. The following query determines the number of film rentals for each customer and sorts the results in descending order:

```
mysql> SELECT customer_id, count(*) num_rentals
    -> FROM rental
    -> GROUP BY customer_id
    -> ORDER BY 2 desc;
+-------------+-------------+
| customer_id | num_rentals |
+-------------+-------------+
|         148 |          46 |
|         526 |          45 |
|         236 |          42 |
```

```
|         144 |          42 |
|          75 |          41 |
|         469 |          40 |
|         197 |          40 |
|         137 |          39 |
|         468 |          39 |
|         178 |          39 |
|         459 |          38 |
|         410 |          38 |
|           5 |          38 |
|         295 |          38 |
|         257 |          37 |
|         366 |          37 |
|         176 |          37 |
|         198 |          37 |
|         267 |          36 |
|         439 |          36 |
|         354 |          36 |
|         348 |          36 |
|         380 |          36 |
|          29 |          36 |
|         371 |          35 |
|         403 |          35 |
|          21 |          35 |
...
|         136 |          15 |
|         248 |          15 |
|         110 |          14 |
|         281 |          14 |
|          61 |          14 |
|         318 |          12 |
+-------------+-------------+
599 rows in set (0.16 sec)
```

Looking at the results, the third and fourth customers in the result set both rented 42 films; should they both receive the same ranking of 3? And if so, should the customer with 41 rentals be given the ranking 4, or should we skip one and assign ranking 5? To see how each function handles ties when assigning rankings, the next query adds three more columns, each one employing a different ranking function:

```
mysql> SELECT customer_id, count(*) num_rentals,
    ->   row_number() over (order by count(*) desc)
```

```
row_number_rnk,
    ->   rank() over (order by count(*) desc) rank_rnk,
    ->   dense_rank() over (order by count(*) desc)
dense_rank_rnk
    -> FROM rental
    -> GROUP BY customer_id
    -> ORDER BY 2 desc;
+-------------+-------------+----------------+----------+---
-------------+
| customer_id | num_rentals | row_number_rnk | rank_rnk |
dense_rank_rnk |
+-------------+-------------+----------------+----------+---
-------------+
|         148 |          46 |              1 |        1
|              1 |
|         526 |          45 |              2 |        2
|              2 |
|         144 |          42 |              3 |        3
|              3 |
|         236 |          42 |              4 |        3
|              3 |
|          75 |          41 |              5 |        5
|              4 |
|         197 |          40 |              6 |        6
|              5 |
|         469 |          40 |              7 |        6
|              5 |
|         468 |          39 |             10 |        8
|              6 |
|         137 |          39 |              8 |        8
|              6 |
|         178 |          39 |              9 |        8
|              6 |
|           5 |          38 |             11 |       11
|              7 |
|         295 |          38 |             12 |       11
|              7 |
|         410 |          38 |             13 |       11
|              7 |
|         459 |          38 |             14 |       11
|              7 |
|         198 |          37 |             16 |       15
|              8 |
|         257 |          37 |             17 |       15
|              8 |
|         366 |          37 |             18 |       15
|              8 |
|         176 |          37 |             15 |       15
```

```
|              8 |
|          348 |           36 |                21 |        19
|              9 |
|          354 |           36 |                22 |        19
|              9 |
|          380 |           36 |                23 |        19
|              9 |
|          439 |           36 |                24 |        19
|              9 |
|           29 |           36 |                19 |        19
|              9 |
|          267 |           36 |                20 |        19
|              9 |
|           50 |           35 |                26 |        25
|             10 |
|          506 |           35 |                37 |        25
|             10 |
|          368 |           35 |                32 |        25
|             10 |
|           91 |           35 |                27 |        25
|             10 |
|          371 |           35 |                33 |        25
|             10 |
|          196 |           35 |                28 |        25
|             10 |
|          373 |           35 |                34 |        25
|             10 |
|          204 |           35 |                29 |        25
|             10 |
|          381 |           35 |                35 |        25
|             10 |
|          273 |           35 |                30 |        25
|             10 |
|           21 |           35 |                25 |        25
|             10 |
|          403 |           35 |                36 |        25
|             10 |
|          274 |           35 |                31 |        25
|             10 |
|           66 |           34 |                42 |        38
|             11 |
...
|          136 |           15 |               594 |       594
|             30 |
|          248 |           15 |               595 |       594
|             30 |
|          110 |           14 |               597 |       596
|             31 |
```

```
|         281 |          14 |             598 |       596
|             31 |
|          61 |          14 |             596 |       596
|             31 |
|         318 |          12 |             599 |       599
|             32 |
+------------+------------+--------------+---------+---
-------------+
599 rows in set (0.01 sec)
```

The third column uses the `row_number` function to assign a unique ranking to each row, without regard to ties. Each of the 599 rows are assigned a number from 1 to 599, with the ranking value arbitrarily assigned for customers having the same number of film rentals. The next two columns, however, assign the same ranking in case of a tie, but the difference lies in whether not a gap is left in the ranking values after a tie. Looking at row 5 of the result set, you can see that the `rank` function skips the value 4 and assigns the value 5, whereas the `dense_rank` function assigns the value 4.

To get back to the original request, how would you identify the top 10 customers? There are 3 possible solutions:

- Use the `row_number` function to identify customers ranked from 1 to 10, which results in exactly 10 customers in this example, but in other cases might exclude customers having the same number of rentals as the 10th ranked customer.
- Use the `rank` function to identify customers ranked 10 or less, which also results in exactly 10 customers.
- Use the `dense_rank` function to identify customers ranked 10 or less, which yields a list of 37 customers.

If there are no ties in your result set, then any of these functions will suffice, but for many situations the `rank` function may be the best option.

Generating Multiple Rankings

The example in the previous section generates a single ranking across the entire set of customers, but what if you want to generate multiple sets of rankings within the same result set? To extend the prior example, let's say the marketing department decides to offer free film rentals to the top 5 customers every month. To generate the data, the `rental_month` column can be added to the previous query:

```
mysql> SELECT customer_id,
    ->   monthname(rental_date) rental_month,
    ->   count(*) num_rentals
    -> FROM rental
    -> GROUP BY customer_id, monthname(rental_date)
    -> ORDER BY 2, 3 desc;
+-------------+--------------+-------------+
| customer_id | rental_month | num_rentals |
+-------------+--------------+-------------+
|         119 | August       |          18 |
|          15 | August       |          18 |
|         569 | August       |          18 |
|         148 | August       |          18 |
|         141 | August       |          17 |
|          21 | August       |          17 |
|         266 | August       |          17 |
|         418 | August       |          17 |
|         410 | August       |          17 |
|         342 | August       |          17 |
|         274 | August       |          16 |
...
|         281 | August       |           2 |
|         318 | August       |           1 |
|          75 | February     |           3 |
|         155 | February     |           2 |
|         175 | February     |           2 |
|         516 | February     |           2 |
```

```
|         361 | February     |           2 |
|         269 | February     |           2 |
|         208 | February     |           2 |
|          53 | February     |           2 |
...
|          22 | February     |           1 |
|         472 | February     |           1 |
|         148 | July         |          22 |
|         102 | July         |          21 |
|         236 | July         |          20 |
|          75 | July         |          20 |
|          91 | July         |          19 |
|          30 | July         |          19 |
|          64 | July         |          19 |
|         137 | July         |          19 |
...
|         339 | May          |           1 |
|         485 | May          |           1 |
|         116 | May          |           1 |
|         497 | May          |           1 |
|         180 | May          |           1 |
+-------------+--------------+-------------+
2466 rows in set (0.02 sec)
```

In order to create a new set of rankings for each month, you will need to add something to the `rank` function in order to describe how to divide the result set into different data windows (months, in this case). This is done using the `partition by` clause, which is added to the `over` clause:

```
mysql> SELECT customer_id,
    ->   monthname(rental_date) rental_month,
    ->   count(*) num_rentals,
    ->   rank() over (partition by monthname(rental_date)
    ->     order by count(*) desc) rank_rnk
    -> FROM rental
    -> GROUP BY customer_id, monthname(rental_date)
    -> ORDER BY 2, 3 desc;
+-------------+--------------+-------------+----------+
| customer_id | rental_month | num_rentals | rank_rnk |
+-------------+--------------+-------------+----------+
|         569 | August       |          18 |        1 |
|         119 | August       |          18 |        1 |
```

```
|          148 | August      |          18 |        1 |
|           15 | August      |          18 |        1 |
|          141 | August      |          17 |        5 |
|          410 | August      |          17 |        5 |
|          418 | August      |          17 |        5 |
|           21 | August      |          17 |        5 |
|          266 | August      |          17 |        5 |
|          342 | August      |          17 |        5 |
|          144 | August      |          16 |       11 |
|          274 | August      |          16 |       11 |
...
|          164 | August      |           2 |      596 |
|          318 | August      |           1 |      599 |
|           75 | February    |           3 |        1 |
|          457 | February    |           2 |        2 |
|           53 | February    |           2 |        2 |
|          354 | February    |           2 |        2 |

|          352 | February    |           1 |       24 |
|          373 | February    |           1 |       24 |
|          148 | July        |          22 |        1 |
|          102 | July        |          21 |        2 |
|          236 | July        |          20 |        3 |
|           75 | July        |          20 |        3 |
|           91 | July        |          19 |        5 |
|          354 | July        |          19 |        5 |
|           30 | July        |          19 |        5 |
|           64 | July        |          19 |        5 |
|          137 | July        |          19 |        5 |
|          526 | July        |          19 |        5 |
|          366 | July        |          19 |        5 |
|          595 | July        |          19 |        5 |
|          469 | July        |          18 |       13 |
...
|          457 | May         |           1 |      347 |
|          356 | May         |           1 |      347 |
|          481 | May         |           1 |      347 |
|           10 | May         |           1 |      347 |
+-------------+-------------+-------------+----------+
2466 rows in set (0.03 sec)
```

Looking at the results, you can see that the rankings are reset to 1 for each month. In order to generate the desired results for the marketing department (top 5 customers from each month), you can simply wrap the previous query in a subquery, and add a filter condition to exclude any rows with a ranking higher than 5:

```
SELECT customer_id, rental_month, num_rentals,
  rank_rnk ranking
FROM
 (SELECT customer_id,
    monthname(rental_date) rental_month,
    count(*) num_rentals,
    rank() over (partition by monthname(rental_date)
      order by count(*) desc) rank_rnk
  FROM rental
  GROUP BY customer_id, monthname(rental_date)
 ) cust_rankings
WHERE rank_rnk <= 5
ORDER BY rental_month, num_rentals desc, rank_rnk;
```

Since analytic functions can only be used in the `SELECT` clause, you will often need to nest queries if you need to do any filtering or grouping based on the results from the analytic function.

Reporting Functions

Along with generating rankings, another common use for analytic functions is to find outliers (e.g. min or max values) or to generate sums or averages across an entire data set. For these types of uses, you will be using aggregate functions (`min` , `max`, `avg`, `sum`, `count`), but instead of using them with a `group by` clause, you will pair them with an `over` clause. Here's an example that generates monthly and grand totals for all payments of $10 or higher:

```
mysql> SELECT monthname(payment_date) payment_month,
    ->   amount,
    ->   sum(amount)
    ->     over (partition by monthname(payment_date))
monthly_total,
```

```
    ->   sum(amount) over () grand_total
    -> FROM payment
    -> WHERE amount >= 10
    -> ORDER BY 1;
+---------------+--------+---------------+-------------+
| payment_month | amount | monthly_total | grand_total |
+---------------+--------+---------------+-------------+
| August        |  10.99 |        521.53 |     1262.86 |
| August        |  11.99 |        521.53 |     1262.86 |
| August        |  10.99 |        521.53 |     1262.86 |
| August        |  10.99 |        521.53 |     1262.86 |
...
| August        |  10.99 |        521.53 |     1262.86 |
| August        |  10.99 |        521.53 |     1262.86 |
| August        |  10.99 |        521.53 |     1262.86 |
| July          |  10.99 |        519.53 |     1262.86 |
| July          |  10.99 |        519.53 |     1262.86 |
| July          |  10.99 |        519.53 |     1262.86 |
| July          |  10.99 |        519.53 |     1262.86 |
...
| July          |  10.99 |        519.53 |     1262.86 |
| July          |  10.99 |        519.53 |     1262.86 |
| July          |  10.99 |        519.53 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  11.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| June          |  10.99 |        165.85 |     1262.86 |
| May           |  10.99 |         55.95 |     1262.86 |
| May           |  10.99 |         55.95 |     1262.86 |
| May           |  10.99 |         55.95 |     1262.86 |
| May           |  10.99 |         55.95 |     1262.86 |
| May           |  11.99 |         55.95 |     1262.86 |
+---------------+--------+---------------+-------------+
114 rows in set (0.01 sec)
```

The `grand_total` column contains the same value ($1,262.86) for every row because the `over` clause is empty, which specifies that the summation be done over the entire result set. The `monthly_total` column, however, contains a different value for each month, since there is a `partition by` clause specifying that the result set be split into multiple data windows (one for each month).

While it may seem of little value to include a column such as `grand_total` having the same value for every row, these types of columns can also be used for calculations, as shown by the next query:

```
mysql> SELECT monthname(payment_date) payment_month,
    ->   sum(amount) month_total,
    ->   round(sum(amount) / sum(sum(amount)) over ()
    ->     * 100, 2) pct_of_total
    -> FROM payment
    -> GROUP BY monthname(payment_date);
+---------------+-------------+--------------+
| payment_month | month_total | pct_of_total |
+---------------+-------------+--------------+
| May           |     4824.43 |         7.16 |
| June          |     9631.88 |        14.29 |
| July          |    28373.89 |        42.09 |
| August        |    24072.13 |        35.71 |
| February      |      514.18 |         0.76 |
+---------------+-------------+--------------+
5 rows in set (0.04 sec)
```

This query calculates the total payments for each month by summing the `amount` column, and then calculates the percentage of the total payments for each month by summing the monthly sums to use as the denominator in the calculation.

Reporting functions may also be used for comparisons, such as the next query, which uses a `case` expression to determine whether a monthly total is the max, min, or somewhere in the middle:

```
mysql> SELECT monthname(payment_date) payment_month,
    ->   sum(amount) month_total,
    ->   CASE sum(amount)
    ->     WHEN max(sum(amount)) over () THEN 'Highest'
    ->     WHEN min(sum(amount)) over () THEN 'Lowest'
    ->     ELSE 'Middle'
    ->   END descriptor
    -> FROM payment
    -> GROUP BY monthname(payment_date);
+---------------+-------------+------------+
| payment_month | month_total | descriptor |
+---------------+-------------+------------+
| May           |     4824.43 | Middle     |
| June          |     9631.88 | Middle     |
| July          |    28373.89 | Highest    |
| August        |    24072.13 | Middle     |
| February      |      514.18 | Lowest     |
+---------------+-------------+------------+
5 rows in set (0.04 sec)
```

The `descriptor` column acts as a quasi-ranking function, in that it helps identify the top/bottom values across a set of rows.

Window Frames

As described earlier in the chapter, data windows for analytic functions are defined using the `partition by` clause, which allows you to group rows by common values. But what if you need even finer control over which rows to include in a data window? For example, perhaps you want to generate a running total starting from the beginning of the year and up to the current row. For these types of calculations, you can include a “frame” sub clause to define exactly which rows to include in the data window. Here’s a query that sums payments for each week, and includes a reporting function to calculate the rolling sum:

```
mysql> SELECT yearweek(payment_date) payment_week,
    ->   sum(amount) week_total,
    ->   sum(sum(amount))
    ->     over (order by yearweek(payment_date)
    ->       rows unbounded preceding) rolling_sum
    -> FROM payment
    -> GROUP BY yearweek(payment_date)
    -> ORDER BY 1;
+--------------+------------+-------------+
| payment_week | week_total | rolling_sum |
+--------------+------------+-------------+
|       200521 |    2847.18 |     2847.18 |
|       200522 |    1977.25 |     4824.43 |
|       200524 |    5605.42 |    10429.85 |
|       200525 |    4026.46 |    14456.31 |
|       200527 |    8490.83 |    22947.14 |
|       200528 |    5983.63 |    28930.77 |
|       200530 |   11031.22 |    39961.99 |
|       200531 |    8412.07 |    48374.06 |
|       200533 |   10619.11 |    58993.17 |
|       200534 |    7909.16 |    66902.33 |
|       200607 |     514.18 |    67416.51 |
+--------------+------------+-------------+
11 rows in set (0.04 sec)
```

The `rolling_sum` column expression includes the `rows unbounded preceding` sub clause to define a data window from the beginning of the result set up to and including the current row. The data window consists of a single row for the first row in the result set, two rows for the second row, etc. The value for the last row is the summation of the entire result set.

Along with rolling sums, you can calculate rolling averages. Here’s a query that calculates a 3-week rolling average of total payments:

```
mysql> SELECT yearweek(payment_date) payment_week,
    ->   sum(amount) week_total,
    ->   avg(sum(amount))
    ->     over (order by yearweek(payment_date)
    ->       rows between 1 preceding and 1 following)
rolling_3wk_avg
    -> FROM payment
    -> GROUP BY yearweek(payment_date)
    -> ORDER BY 1;
+--------------+------------+-----------------+
| payment_week | week_total | rolling_3wk_avg |
+--------------+------------+-----------------+
|       200521 |    2847.18 |     2412.215000 |
|       200522 |    1977.25 |     3476.616667 |
|       200524 |    5605.42 |     3869.710000 |
|       200525 |    4026.46 |     6040.903333 |
|       200527 |    8490.83 |     6166.973333 |
|       200528 |    5983.63 |     8501.893333 |
|       200530 |   11031.22 |     8475.640000 |
|       200531 |    8412.07 |    10020.800000 |
|       200533 |   10619.11 |     8980.113333 |
|       200534 |    7909.16 |     6347.483333 |
|       200607 |     514.18 |     4211.670000 |
+--------------+------------+-----------------+
11 rows in set (0.03 sec)
```

The `rolling_3wk_avg` column defines a data window consisting of the current row, the prior row, and the next row. The data window will therefore consist of 3 rows, except for the first and last rows, which will have a data window consisting of just 2 rows (since there is no prior row for the first row, and no next row for the last row).

Specifying a number of rows for your data window works fine in many cases, but if there are gaps in your data you might want to try a different approach. In the previous result set, for example, there is data for weeks 200521, 200522, and 200524, but no date for week

200523. If you want to specify a date interval rather than a number of rows, you can specify a **range** for your data window, as shown in the next query:

```
mysql> SELECT date(payment_date), sum(amount),
    ->   avg(sum(amount)) over (order by date(payment_date)
    ->     range between interval 3 day preceding
    ->       and interval 3 day following) 7_day_avg
    -> FROM payment
    -> WHERE payment_date BETWEEN '2005-07-01' AND '2005-09-
01'
    -> GROUP BY date(payment_date)
    -> ORDER BY 1;
+--------------------+-------------+-------------+
| date(payment_date) | sum(amount) | 7_day_avg   |
+--------------------+-------------+-------------+
| 2005-07-05         |      128.73 | 1603.740000 |
| 2005-07-06         |     2131.96 | 1698.166000 |
| 2005-07-07         |     1943.39 | 1738.338333 |
| 2005-07-08         |     2210.88 | 1766.917143 |
| 2005-07-09         |     2075.87 | 2049.390000 |
| 2005-07-10         |     1939.20 | 2035.628333 |
| 2005-07-11         |     1938.39 | 2054.076000 |
| 2005-07-12         |     2106.04 | 2014.875000 |
| 2005-07-26         |      160.67 | 2046.642500 |
| 2005-07-27         |     2726.51 | 2206.244000 |
| 2005-07-28         |     2577.80 | 2316.571667 |
| 2005-07-29         |     2721.59 | 2388.102857 |
| 2005-07-30         |     2844.65 | 2754.660000 |
| 2005-07-31         |     2868.21 | 2759.351667 |
| 2005-08-01         |     2817.29 | 2795.662000 |
| 2005-08-02         |     2726.57 | 2814.180000 |
| 2005-08-16         |      111.77 | 1973.837500 |
| 2005-08-17         |     2457.07 | 2123.822000 |
| 2005-08-18         |     2710.79 | 2238.086667 |
| 2005-08-19         |     2615.72 | 2286.465714 |
| 2005-08-20         |     2723.76 | 2630.928571 |
| 2005-08-21         |     2809.41 | 2659.905000 |
| 2005-08-22         |     2576.74 | 2649.728000 |
| 2005-08-23         |     2523.01 | 2658.230000 |
+--------------------+-------------+-------------+
24 rows in set (0.03 sec)
```

The `7_day_avg` column specifies a range of +/- 3 days and will include only those rows whose `payment_date` values fall within that range. For the 2005-08-16 calculation, for example, only the values for 08-16, 08-17, 08-19, and 08-20 are included, since there are no rows for the 3 prior dates (08-13 through 08-15).

Lag and Lead

Along with computing sums and averages over a data window, another common reporting task involves comparing values from one row to another. For example, if you are generating monthly sales totals, you may be asked to create a column showing the percentage difference from the prior month, which will require a way to retrieve the monthly sales total from the previous row. This can be accomplished using the `lag` function, which will retrieve a column value from a prior row in the result set, or the `lead` function, which will retrieve a column value from a following row. Here's an example using both functions:

```
mysql> SELECT yearweek(payment_date) payment_week,
    ->   sum(amount) week_total,
    ->   lag(sum(amount), 1)
    ->     over (order by yearweek(payment_date))
prev_wk_tot,
    ->   lead(sum(amount), 1)
    ->     over (order by yearweek(payment_date))
next_wk_tot
    -> FROM payment
    -> GROUP BY yearweek(payment_date)
    -> ORDER BY 1;
+--------------+------------+-------------+-------------+
| payment_week | week_total | prev_wk_tot | next_wk_tot |
+--------------+------------+-------------+-------------+
|       200521 |    2847.18 |        NULL |     1977.25 |
|       200522 |    1977.25 |     2847.18 |     5605.42 |
|       200524 |    5605.42 |     1977.25 |     4026.46 |
```

```
|        200525 |    4026.46 |     5605.42 |     8490.83 |
|        200527 |    8490.83 |     4026.46 |     5983.63 |
|        200528 |    5983.63 |     8490.83 |    11031.22 |
|        200530 |   11031.22 |     5983.63 |     8412.07 |
|        200531 |    8412.07 |    11031.22 |    10619.11 |
|        200533 |   10619.11 |     8412.07 |     7909.16 |
|        200534 |    7909.16 |    10619.11 |      514.18 |
|        200607 |     514.18 |     7909.16 |        NULL |
+--------------+------------+-------------+-------------+
11 rows in set (0.03 sec)
```

Looking at the results, , the weekly total of 8,490.43 for week 200527 also appears in the `next_wk_tot` column for week 200525, as well as in the `prev_wk_tot` column for week 200528. Since there is no row prior to 200521 in the result set, the value generated by the `lag` function is NULL for the first row; likewise, the value generated by the `lead` function is NULL for the last row in the result set. Both `lag` and `lead` allow for an optional 2nd parameter (which defaults to 1) to describe the number of rows prior/following from which to retrieve the column value.

Here's how you could use the `lag` function to generate the percentage difference from the prior week:

```
mysql> SELECT yearweek(payment_date) payment_week,
    ->   sum(amount) week_total,
    ->   round((sum(amount) - lag(sum(amount), 1)
    ->     over (order by yearweek(payment_date)))
    ->     / lag(sum(amount), 1)
    ->       over (order by yearweek(payment_date))
    ->     * 100, 1) pct_diff
    -> FROM payment
    -> GROUP BY yearweek(payment_date)
    -> ORDER BY 1;
+--------------+------------+----------+
```

```
| payment_week | week_total | pct_diff |
+--------------+------------+----------+
|       200521 |    2847.18 |     NULL |
|       200522 |    1977.25 |    -30.6 |
|       200524 |    5605.42 |    183.5 |
|       200525 |    4026.46 |    -28.2 |
|       200527 |    8490.83 |    110.9 |
|       200528 |    5983.63 |    -29.5 |
|       200530 |   11031.22 |     84.4 |
|       200531 |    8412.07 |    -23.7 |
|       200533 |   10619.11 |     26.2 |
|       200534 |    7909.16 |    -25.5 |
|       200607 |     514.18 |    -93.5 |
+--------------+------------+----------+
11 rows in set (0.07 sec)
```

Comparing values from different rows in the same result set is a common practice in reporting systems, so you will likely find many uses for the `lag` and `lead` functions.

Test Your Knowledge

The following exercises are designed to test your understanding of analytic functions. When you're finished, please see Appendix C.

For all exercises in this section, use the following data set from the Sales_Fact table:

```
Sales_Fact
+---------+----------+-----------+
| year_no | month_no | tot_sales |
+---------+----------+-----------+
|    2019 |        1 |     19228 |
|    2019 |        2 |     18554 |
|    2019 |        3 |     17325 |
```

```
|    2019 |         4 |     13221 |
|    2019 |         5 |      9964 |
|    2019 |         6 |     12658 |
|    2019 |         7 |     14233 |
|    2019 |         8 |     17342 |
|    2019 |         9 |     16853 |
|    2019 |        10 |     17121 |
|    2019 |        11 |     19095 |
|    2019 |        12 |     21436 |
|    2020 |         1 |     20347 |
|    2020 |         2 |     17434 |
|    2020 |         3 |     16225 |
|    2020 |         4 |     13853 |
|    2020 |         5 |     14589 |
|    2020 |         6 |     13248 |
|    2020 |         7 |      8728 |
|    2020 |         8 |      9378 |
|    2020 |         9 |     11467 |
|    2020 |        10 |     13842 |
|    2020 |        11 |     15742 |
|    2020 |        12 |     18636 |
+---------+---------+-----------+
24 rows in set (0.00 sec)
```

Exercise 16-1

Write a query that retrieves every row from Sales_Fact, and add a column to generate a ranking based on the tot_sales column values. The highest value should receive a ranking of 1, and the lowest a ranking of 24.

Exercise 16-2

Modify the query from exercise 16-1 to generate two sets of rankings from 1 to 12, one for 2019 data and one for 2020.

Exercise 16-3

Write a query that retrieves all 2020 data, and include a column which will contain the tot_sales value from the previous month.

Chapter 17. Working with Large Databases

In the early days of relational databases, hard drive capacity was measured in megabytes, and databases were generally easy to administer simply because they couldn’t get very large. Today, however, hard drive capacity has ballooned to 15TB, a modern disk array can store over 4PB of data, and storage in the cloud is essentially limitless. At some point along this journey, individual database tables started to become unwieldy, with row counts into the billions. Some databases have become too big to fit on even the largest database appliances, causing designers to find ways to spread data across many different databases, or to look to alternate technologies for data storage. This chapter looks at some of the strategies which have evolved for handling very large databases.

Partitioning

When exactly does a database table become “too big”? If you ask this question to 10 different data architects/administrators/developers, you will likely get 10 different answers. Most people, however, would agree that the following tasks become more difficult and/or time consuming as a table grows past a few million rows:

- Query execution requiring full table scans

- Index creation/rebuild
- Data archival/deletion
- Generation of table/index statistics
- Table relocation (e.g. move to a different tablespace)
- Database backups

These tasks can start as routine when a database is small, then become time consuming as more and more data accumulates, and then become problematic/impossible due to limited administrative time windows. The best way to prevent administrative issues from occurring in the future is to break large tables into pieces, or partitions, when the table is first created (although tables can be partitioned later, it is easier to do so initially). Administrative tasks can be performed on individual partitions, often in parallel, and some tasks can skip one or more partitions entirely.

Partitioning Concepts

Table partitioning was introduced in the late 1990's by Oracle, but since then every major database server has added the ability to partition tables and indexes. When a table is partitioned, two or more table partitions are created, each having the exact same definition, but with non-overlapping subsets of data. For example, a table containing sales data could be partitioned by month using the column containing the sale date, or it could be partitioned by geographic region using the state/province code.

Once a table has been partitioned, the table itself becomes a virtual concept; the partitions hold the data, and any indexes are built on the data in the partitions. However, the database users can still interact with the table without knowing that the table had been partitioned. This is similar in concept to a view, in that the users interact with schema objects which are interfaces rather than actual tables. While every partition must have the same schema definition (columns, column types, etc.), there are several administrative features which can differ for each partition:

- Partitions may be stored on different tablespaces, which can be on different physical storage tiers
- Partitions can be compressed using different compression schemes
- Local indexes (more on this shortly) can be dropped for some partitions
- Table statistics can be frozen on some partitions, while being periodically refreshed on others
- Individual partitions can be pinned into memory, or stored in the database's flash storage tier

Thus, table partitioning allows for flexibility with data storage and administration, while still presenting the simplicity of a single table to your user community.

Table Partitioning

The partitioning scheme available in most relational databases is *horizontal partitioning*, which assigns entire rows to exactly one partition. Tables may also be partitioned *vertically*, which involves assigning sets of columns to different partitions, but this must be done manually. When partitioning a table horizontally, you must choose a *partition key*, which is the column whose values are used to assign a row to a particular partition. In most cases, a table's partition key consists of a single column, and a *partitioning function* is applied to this column to determine in which partition each row should reside.

Index Partitioning

If your partitioned table has indexes, you will get to choose whether a particular index should stay intact, known as a *global index*, or be broken into pieces such that each partition has its own index, which is called a *local index*. Global indexes span all partitions of the table and are useful for queries which do not specify a value for the partition key. For example, let's say your table is partitioned on the `sale_date` column, and a user executes the following query:

```
SELECT sum(amount) FROM sales WHERE geo_region_cd = 'US'
```

Since this query does not include a filter condition on the `sale_date` column, the server will need to search every partition in order to find the total US sales. If a global index is built on the `geo_region_cd` column, however, then the server could use this index to quickly find all of the rows containing US sales.

Partitioning Methods

While each database server has their own unique partitioning features, the next three sections describe the common partitioning methods available across most servers.

Range Partitioning

Range partitioning was the first partitioning method to be implemented, and it is still one of the most-widely used. While range partitioning can be used for several different column types, the most common usage is to break up tables by date ranges. For example, a table named `sales` could be partitioned using the `sale_date` column such that data for each week is stored in a different partition:

```
mysql> CREATE TABLE sales
    ->  (sale_id INT NOT NULL,
    ->   cust_id INT NOT NULL,
    ->   store_id INT NOT NULL,
    ->   sale_date DATE NOT NULL,
    ->   amount DECIMAL(9,2)
    ->  )
    -> PARTITION BY RANGE (yearweek(sale_date))
    ->  (PARTITION s1 VALUES LESS THAN (202002),
    ->   PARTITION s2 VALUES LESS THAN (202003),
    ->   PARTITION s3 VALUES LESS THAN (202004),
    ->   PARTITION s4 VALUES LESS THAN (202005),
    ->   PARTITION s5 VALUES LESS THAN (202006),
    ->   PARTITION s999 VALUES LESS THAN (MAXVALUE)
    ->  );
Query OK, 0 rows affected (1.78 sec)
```

This statement creates six different partitions; one for each of the first five weeks of 2020, and a sixth partition named s999 to hold any rows beyond week 5 of year 2020. For this table, the `yearweek(sale_date)` expression is used as the partitioning function, and the `sale_date` column serves as the partitioning key. To see the metadata about your partitioned tables, you can use the `partitions` table in the `information_schema` database:

```
mysql> SELECT partition_name, partition_method,
partition_expression
    -> FROM information_schema.partitions
    -> WHERE table_name = 'sales'
    -> ORDER BY partition_ordinal_position;
```

```
+----------------+------------------+----------------------
--+
| PARTITION_NAME | PARTITION_METHOD |
PARTITION_EXPRESSION    |
+----------------+------------------+----------------------
--+
| s1             | RANGE            |
yearweek(`sale_date`,0) |
| s2             | RANGE            |
yearweek(`sale_date`,0) |
| s3             | RANGE            |
yearweek(`sale_date`,0) |
| s4             | RANGE            |
yearweek(`sale_date`,0) |
| s5             | RANGE            |
yearweek(`sale_date`,0) |
| s999           | RANGE            |
yearweek(`sale_date`,0) |
+----------------+------------------+----------------------
--+
6 rows in set (0.00 sec)
```

One of the administrative tasks which will need to be performed on the `sales` table involves generating new partitions to hold future data (to keep data from being added to the MAXVALUE partition). Different databases handle this in different ways, but in MySQL you could use the `reorganize partition` clause of the `alter table` command to split the s999 partition into three pieces:

```
ALTER TABLE sales REORGANIZE PARTITION s999 INTO
 (PARTITION s6 VALUES LESS THAN (202007),
  PARTITION s7 VALUES LESS THAN (202008),
  PARTITION s999 VALUES LESS THAN (MAXVALUE)
 );
```

If you execute the previous metadata query again, you will now see 8 partitions:

```
mysql> SELECT partition_name, partition_method, partition_e
    -> FROM information_schema.partitions
    -> WHERE table_name = 'sales'
    -> ORDER BY partition_ordinal_position;
+----------------+------------------+----------------------
| PARTITION_NAME | PARTITION_METHOD | PARTITION_EXPRESSION
+----------------+------------------+----------------------
| s1             | RANGE            | yearweek(`sale_date`,
| s2             | RANGE            | yearweek(`sale_date`,
| s3             | RANGE            | yearweek(`sale_date`,
| s4             | RANGE            | yearweek(`sale_date`,
| s5             | RANGE            | yearweek(`sale_date`,
| s6             | RANGE            | yearweek(`sale_date`,
| s7             | RANGE            | yearweek(`sale_date`,
| s999           | RANGE            | yearweek(`sale_date`,
+----------------+------------------+----------------------
8 rows in set (0.00 sec)
```

Next, let's add a couple of rows to the table:

```
mysql> INSERT INTO sales
    -> VALUES
    ->  (1, 1, 1, '2020-01-18', 2765.15),
    ->  (2, 3, 4, '2020-02-07', 5322.08);
Query OK, 2 rows affected (0.18 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

The table now has 2 rows, but into which partitions where they inserted? To find out, let's use the `partition` sub clause of the `from` clause to count the number of rows in each partition:

```
mysql> SELECT concat('# of rows in S1 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s1) UNION ALL
    -> SELECT concat('# of rows in S2 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s2) UNION ALL
```

```
    -> SELECT concat('# of rows in S3 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s3) UNION ALL
    -> SELECT concat('# of rows in S4 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s4) UNION ALL
    -> SELECT concat('# of rows in S5 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s5) UNION ALL
    -> SELECT concat('# of rows in S6 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s6) UNION ALL
    -> SELECT concat('# of rows in S7 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s7) UNION ALL
    -> SELECT concat('# of rows in S999 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (s999);
+-----------------------+
| partition_rowcount    |
+-----------------------+
| # of rows in S1 = 0   |
| # of rows in S2 = 1   |
| # of rows in S3 = 0   |
| # of rows in S4 = 0   |
| # of rows in S5 = 1   |
| # of rows in S6 = 0   |
| # of rows in S7 = 0   |
| # of rows in S999 = 0 |
+-----------------------+
8 rows in set (0.00 sec)
```

The results show that one row was inserted into partition S2, and the other row was inserted into the S5 partition. The ability to query a specific partition involves having knowledge of the partitioning scheme, so it is unlikely that your user community will be executing these types of queries, but it is commonly used for administrative activities.

List Partitioning

If the column chosen as the partitioning key contains state codes (e.g. CA, TX, VA, etc.), currencies (e.g. USD, EUR, JPY, etc.), or some other enumerated set of values, you may want to utilize list partitioning, which allows you to specify which values will be assigned to each partition. For example, let's say that the `sales` table includes the column `geo_region_cd`, which contains the following values:

```
+---------------+--------------------------+
| geo_region_cd | description              |
+---------------+--------------------------+
| US_NE         | United States North East |
| US_SE         | United States South East |
| US_MW         | United States Mid West   |
| US_NW         | United States North West |
| US_SW         | United States South West |
| CAN           | Canada                   |
| MEX           | Mexico                   |
| EUR_E         | Eastern Europe           |
| EUR_W         | Western Europe           |
| CHN           | China                    |
| JPN           | Japan                    |
| IND           | India                    |
| KOR           | Korea                    |
+---------------+--------------------------+
13 rows in set (0.00 sec)
```

You could group these values into geographic regions, and create a partition for each one, as in:

```
mysql> CREATE TABLE sales
    ->  (sale_id INT NOT NULL,
    ->   cust_id INT NOT NULL,
    ->   store_id INT NOT NULL,
    ->   sale_date DATE NOT NULL,
    ->   geo_region_cd VARCHAR(6) NOT NULL,
    ->   amount DECIMAL(9,2)
    ->  )
    -> PARTITION BY LIST COLUMNS (geo_region_cd)
    ->  (PARTITION NORTHAMERICA VALUES IN
```

```
('US_NE','US_SE','US_MW',
    ->
'US_NW','US_SW','CAN','MEX'),
    ->   PARTITION EUROPE VALUES IN ('EUR_E','EUR_W'),
    ->   PARTITION ASIA VALUES IN ('CHN','JPN','IND')
    ->  );
Query OK, 0 rows affected (1.13 sec)
```

The table has three partitions, where each partition includes a set of two or more `geo_region_cd` values. Next, let's add a few rows to the table:

```
mysql> INSERT INTO sales
    -> VALUES
    ->  (1, 1, 1, '2020-01-18', 'US_NE', 2765.15),
    ->  (2, 3, 4, '2020-02-07', 'CAN', 5322.08),
    ->  (3, 6, 27, '2020-03-11', 'KOR', 4267.12);
ERROR 1526 (HY000): Table has no partition for value from
column_list
```

It looks like there was a problem, and the error message indicates that one of the geographic region codes was not assigned to a partition. Looking at the `create table` statement, I see that I forgot to add Korea to the ASIA partition. This can be fixed using `alter table` statement:

```
mysql> ALTER TABLE sales REORGANIZE PARTITION ASIA INTO
    ->  (PARTITION ASIA VALUES IN ('CHN','JPN','IND',
'KOR'));
Query OK, 0 rows affected (1.28 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

That seemed to do the trick, but let's check the metadata just to be sure:

```
mysql> SELECT partition_name, partition_expression,
    ->   partition_description
    -> FROM information_schema.partitions
```

```
    -> WHERE table_name = 'sales'
    -> ORDER BY partition_ordinal_position;
+----------------+---------------------+-------------------
--------------+
| PARTITION_NAME | PARTITION_EXPRESSION |
PARTITION_DESCRIPTION           |
+----------------+---------------------+-------------------
--------------+
| NORTHAMERICA   | `geo_region_cd`      |
'US_NE','US_SE','US_MW','US_NW',|
|                |                      |
'US_SW','CAN','MEX'             |
| EUROPE         | `geo_region_cd`      |
'EUR_E','EUR_W'                 |
| ASIA           | `geo_region_cd`      |
'CHN','JPN','IND','KOR'         |
+----------------+---------------------+-------------------
--------------+
3 rows in set (0.00 sec)
```

Korea has indeed been added to the ASIA partition, and the data insertion will now proceed without any issues:

```
mysql> INSERT INTO sales
    -> VALUES
    ->  (1, 1, 1, '2020-01-18', 'US_NE', 2765.15),
    ->  (2, 3, 4, '2020-02-07', 'CAN', 5322.08),
    ->  (3, 6, 27, '2020-03-11', 'KOR', 4267.12);
Query OK, 3 rows affected (0.26 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

While range partitioning allows for a “maxvalue” partition to catch any rows which don’t map to any other partition, it’s important to keep in mind that list partitioning doesn’t provide for a spillover partition. Thus, any time you need to add another column value (e.g. the company starts selling products in Australia), you will need to modify the partitioning definition before rows with the new value can be added to the table.

Hash Partitioning

If your partition key column doesn't lend itself to range or list partitioning, there is a third option which endeavors to distribute rows evenly across a set of partitions. The server does this by applying a *hashing function* to the column value, and this type of partitioning is (not surprisingly) called hash partitioning. Unlike list partitioning, where the column chosen as the partitioning key should only contain a small number of values, hash partitioning works best when the partitioning key column contains a large number of distinct values. Here's another version of the `sales` table, but with 4 hash partitions generated by hashing the values in the `cust_id` column:

```
mysql> CREATE TABLE sales
    ->  (sale_id INT NOT NULL,
    ->   cust_id INT NOT NULL,
    ->   store_id INT NOT NULL,
    ->   sale_date DATE NOT NULL,
    ->   amount DECIMAL(9,2)
    ->  )
    -> PARTITION BY HASH (cust_id)
    ->   PARTITIONS 4
    ->    (PARTITION H1,
    ->     PARTITION H2,
    ->     PARTITION H3,
    ->     PARTITION H4
    ->    );
Query OK, 0 rows affected (1.50 sec)
```

When rows are added to the `sales` table, they will be evenly distributed across the four partitions, which I named H1, H2, H3, and H4. In order to see how good a job it does, let's add 16 rows, each with a different value for the `cust_id` column:

```
mysql> INSERT INTO sales
    -> VALUES
```

```
    ->  (1, 1, 1, '2020-01-18', 1.1), (2, 3, 4, '2020-02-
07', 1.2),
    ->  (3, 17, 5, '2020-01-19', 1.3), (4, 23, 2, '2020-02-
08', 1.4),
    ->  (5, 56, 1, '2020-01-20', 1.6), (6, 77, 5, '2020-02-
09', 1.7),
    ->  (7, 122, 4, '2020-01-21', 1.8), (8, 153, 1, '2020-
02-10', 1.9),
    ->  (9, 179, 5, '2020-01-22', 2.0), (10, 244, 2, '2020-
02-11', 2.1),
    ->  (11, 263, 1, '2020-01-23', 2.2), (12, 312, 4, '2020-
02-12', 2.3),
    ->  (13, 346, 2, '2020-01-24', 2.4), (14, 389, 3, '2020-
02-13', 2.5),
    ->  (15, 472, 1, '2020-01-25', 2.6), (16, 502, 1, '2020-
02-14', 2.7);
Query OK, 16 rows affected (0.19 sec)
Records: 16  Duplicates: 0  Warnings: 0
```

If the hashing function does a good job of distributing the rows evenly, we should ideally see 4 rows in each of the 4 partitions:

```
mysql> SELECT concat('# of rows in H1 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (h1) UNION ALL
    -> SELECT concat('# of rows in H2 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (h2) UNION ALL
    -> SELECT concat('# of rows in H3 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (h3) UNION ALL
    -> SELECT concat('# of rows in H4 = ', count(*))
partition_rowcount
    -> FROM sales PARTITION (h4);
+---------------------+
| partition_rowcount  |
+---------------------+
| # of rows in H1 = 4 |
| # of rows in H2 = 5 |
| # of rows in H3 = 3 |
| # of rows in H4 = 4 |
+---------------------+
4 rows in set (0.00 sec)
```

Given that only 16 rows were inserted, this is a pretty good distribution, and as the number of rows increases, each partition should contain close to 25% of the rows as long as there are a reasonably large number of distinct values for the `cust_id` column.

Composite Partitioning

If you need finer-grained control of how data is allocated to your partitions, you can employ *composite partitioning*, which allows you to use two different types of partitioning for the same table. With composite partitioning, the first partitioning method defines the partitions, and the second partitioning method defines the *subpartitions*. Here's an example, again using the `sales` table, but utilizing both range and hash partitioning:

```
mysql> CREATE TABLE sales
    ->  (sale_id INT NOT NULL,
    ->   cust_id INT NOT NULL,
    ->   store_id INT NOT NULL,
    ->   sale_date DATE NOT NULL,
    ->   amount DECIMAL(9,2)
    ->  )
    -> PARTITION BY RANGE (yearweek(sale_date))
    -> SUBPARTITION BY HASH (cust_id)
    ->  (PARTITION s1 VALUES LESS THAN (202002)
    ->     (SUBPARTITION s1_h1,
    ->      SUBPARTITION s1_h2,
    ->      SUBPARTITION s1_h3,
    ->      SUBPARTITION s1_h4),
    ->   PARTITION s2 VALUES LESS THAN (202003)
    ->     (SUBPARTITION s2_h1,
    ->      SUBPARTITION s2_h2,
    ->      SUBPARTITION s2_h3,
    ->      SUBPARTITION s2_h4),
    ->   PARTITION s3 VALUES LESS THAN (202004)
    ->     (SUBPARTITION s3_h1,
    ->      SUBPARTITION s3_h2,
    ->      SUBPARTITION s3_h3,
    ->      SUBPARTITION s3_h4),
```

```
    ->   PARTITION s4 VALUES LESS THAN (202005)
    ->     (SUBPARTITION s4_h1,
    ->      SUBPARTITION s4_h2,
    ->      SUBPARTITION s4_h3,
    ->      SUBPARTITION s4_h4),
    ->   PARTITION s5 VALUES LESS THAN (202006)
    ->     (SUBPARTITION s5_h1,
    ->      SUBPARTITION s5_h2,
    ->      SUBPARTITION s5_h3,
    ->      SUBPARTITION s5_h4),
    ->   PARTITION s999 VALUES LESS THAN (MAXVALUE)
    ->     (SUBPARTITION s999_h1,
    ->      SUBPARTITION s999_h2,
    ->      SUBPARTITION s999_h3,
    ->      SUBPARTITION s999_h4)
    ->  );
Query OK, 0 rows affected (9.72 sec)
```

There are 6 partitions, each having 4 subpartitions, for a total of 24 subpartitions. Next, let’s re-insert the 16 rows from the earlier example for hash partitioning:

```
mysql> INSERT INTO sales
    -> VALUES
    ->  (1, 1, 1, '2020-01-18', 1.1), (2, 3, 4, '2020-02-
07', 1.2),
    ->  (3, 17, 5, '2020-01-19', 1.3), (4, 23, 2, '2020-02-
08', 1.4),
    ->  (5, 56, 1, '2020-01-20', 1.6), (6, 77, 5, '2020-02-
09', 1.7),
    ->  (7, 122, 4, '2020-01-21', 1.8), (8, 153, 1, '2020-
02-10', 1.9),
    ->  (9, 179, 5, '2020-01-22', 2.0), (10, 244, 2, '2020-
02-11', 2.1),
    ->  (11, 263, 1, '2020-01-23', 2.2), (12, 312, 4, '2020-
02-12', 2.3),
    ->  (13, 346, 2, '2020-01-24', 2.4), (14, 389, 3, '2020-
02-13', 2.5),
    ->  (15, 472, 1, '2020-01-25', 2.6), (16, 502, 1, '2020-
02-14', 2.7);
Query OK, 16 rows affected (0.22 sec)
Records: 16  Duplicates: 0  Warnings: 0
```

When you query the `sales` table, you can retrieve data from one of the partitions, in which case you retrieve data from the 4 subpartitions associated with the partition:

```
mysql> SELECT *
    -> FROM sales PARTITION (s3);
+---------+---------+----------+------------+--------+
| sale_id | cust_id | store_id | sale_date  | amount |
+---------+---------+----------+------------+--------+
|       5 |      56 |        1 | 2020-01-20 |   1.60 |
|      15 |     472 |        1 | 2020-01-25 |   2.60 |
|       3 |      17 |        5 | 2020-01-19 |   1.30 |
|       7 |     122 |        4 | 2020-01-21 |   1.80 |
|      13 |     346 |        2 | 2020-01-24 |   2.40 |
|       9 |     179 |        5 | 2020-01-22 |   2.00 |
|      11 |     263 |        1 | 2020-01-23 |   2.20 |
+---------+---------+----------+------------+--------+
7 rows in set (0.00 sec)
```

Since the table is subpartitioned, you may also retrieve data from a single subpartition:

```
mysql> SELECT *
    -> FROM sales PARTITION (s3_h3);
+---------+---------+----------+------------+--------+
| sale_id | cust_id | store_id | sale_date  | amount |
+---------+---------+----------+------------+--------+
|       7 |     122 |        4 | 2020-01-21 |   1.80 |
|      13 |     346 |        2 | 2020-01-24 |   2.40 |
+---------+---------+----------+------------+--------+
2 rows in set (0.00 sec)
```

This query retrieves data only from the s3_h3 subpartition of the s3 partition.

Partitioning Benefits

One major advantage to partitioning is that you may only need to interact with as few as one partition, rather than the entire table. For example, if your table is range-partitioned on the `sales_date` column, and you execute a query which includes a filter condition such as `WHERE sales_date BETWEEN '2019-12-01' AND '2020-01-15'` , the server will check the table's metadata to determine which partitions actually need to be included. This concept is called *partition pruning*, and it is one of the biggest advantages of table partitioning.

Similarly, if you execute a query which includes a join to a partitioned table, and the query includes a condition on the partitioning column, the server can exclude any partitions which do not contain data pertinent to the query. This is known as *partition-wise joins*, and it is similar to partition pruning in that only those partitions which contain data needed by the query will be included.

From an administrative standpoint, one of the main benefits to partitioning is the ability to quickly delete data which is no longer needed. For example, financial data may be required to be kept online for seven years; if a table has been partitioned based on transaction dates, any partitions holding data greater than seven years old can be dropped. Another administrative advantage to partitioned tables is the ability to perform updates on multiple partitions simultaneously, which can greatly reduce the time needed to touch every row in a table.

Sharding

Let's say you have been hired as the Data Architect for a new social media company. You are told to expect approximately 1 billion users, each of whom will generate 3.7 messages per day on average, and that the data must be available indefinitely. After performing a few calculations, you determine that you would exhaust the biggest available relational database platform in less than a year. One possibility would be to partition not just individual tables, but the entire database. Known as *sharding*, this approach partitions the data across multiple databases (called *shards*), so it is similar to table partitioning, but on a larger scale and with far more complexity. If you were to employ this strategy for the social media company, you might decide to implement 100 separate databases, each one hosting the data for approximately 10 million users.

Sharding is a complex topic, and since this is an introductory book I will refrain from going into details, but here are a few of the issues which would need to be addressed:

- You will need to choose a *sharding key*, which is the value used to determine to which database to connect.
- While large tables will be divided into pieces, with individual rows assigned to a single shard, smaller reference tables may need to be replicated to all shards, and a strategy needs to be defined for how reference data can be modified and changes propagated to all shards.
- If individual shards become too large (e.g. the social media company now has 2 billion users), you will need a plan for adding more shards and redistributing data across the shards.
- When you need to make schema changes, you will need to have a strategy for deploying the changes across all of the shards so that all schemas stay in synch.

- If application logic needs to access data stored in two or more shards, you need to have a strategy for how to query across multiple databases, and also how to implement transactions across multiple databases.

If this seems complicated, that's because it is, and by the late 2000's many companies began looking for new approaches. The next section looks at other strategies for handling very large data sets, but completely outside the realm of relational databases.

Big Data

After spending some time weighing the pros and cons of sharding, let's say that you (the Data Architect of the social media company) decide to investigate other approaches. Rather than attempting to forge your own path, you might benefit from reviewing the work done by other companies which deal with massive amounts of data: companies like Amazon, Google, Facebook, and Twitter. Together, the set of technologies pioneered by these companies (and others) have been branded as *Big Data*, which has become an industry buzzword but has several possible definitions. However, one way to define the boundaries of Big Data is with the "3 V's":

1. Volume, which in this context generally means billions or trillions of data points
2. Velocity, which is a measure of how quickly data arrives
3. Variety, meaning that data is not always structured (as in rows and columns in a relational database) but can also be unstructured (e.g. emails, videos, photos, audio files, etc.)

So, one way to characterize Big Data is any system designed to handle a huge amount of data of various formats arriving at a rapid pace. The following sections offer a quick description of some of the Big Data technologies which have evolved over the past 15 years or so.

Hadoop

Hadoop is best described as an *ecosystem*, or a set of technologies and tools which work together. Some of the major components include:

- Hadoop Distributed File System (HDFS), which, like the name implies, enables file management across a large number of servers.
- MapReduce, which is a technology used to process large amounts of structured and unstructured data by breaking a task into many small pieces which can be run in parallel across many servers.
- YARN, which is a resource manager and job scheduler for HDFS

Together, these technologies allow for the storage and processing of files across hundreds or even thousands of servers acting as a single logical system. While Hadoop is widely used, querying the data using MapReduce generally requires a programmer, which has led to the development of several SQL interfaces, including Hive, Impala, and Drill.

NoSQL and Document Databases

In a relational database, data must generally conform to a pre-defined schema consisting of tables made up of columns holding numbers, strings, dates, etc. What happens, however, if the structure of the data isn't known beforehand, or if the structure is known but changes frequently? The answer for many companies is to combine both the data and schema definition into documents using a format such as XML (Extensible Markup Language) or JSON (JavaScript Object Notation), and then store the documents in a database. By doing so, various types of data can be stored in the same database without the need to make schema modifications, which makes storage easier but puts the burden on query and analytic tools to make sense of the data stored in the documents.

Document databases are a subset of what are called NoSQL databases, which typically store data using a simple key-value mechanism. For example, using a document database such as MongoDB, you could utilize the User ID as the key to store a JSON document containing all of the customer's data, and other users can read the schema stored within the document to make sense of the data stored within.

Cloud Computing

Prior to the advent of Big Data, most companies had to build their own data centers to house the database, web, and application servers used across the enterprise. With the advent of cloud computing, you can choose to essentially outsource your data center to platforms such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud. One of the biggest benefits to hosting your services in the cloud is instant scalability, which allows you to quickly dial up or down the amount of computing power needed to run your services. Startups love these platforms because they can start writing code without spending any money upfront for servers, storage, networks, or software licenses.

As far as databases are concerned, a quick look at AWS’s database and analytics offerings yields the following options:

- Relational databases (MySQL, Aurora, PostgreSQL, MariaDB, Oracle, and SQL Server)
- In-memory database (ElastiCache)
- Data Warehousing database (Redshift)
- NoSQL database (DynamoDB)
- Document database (DocumentDB)
- Graph database (Neptune)
- Time Series database (TimeStream)
- Hadoop (EMR)
- Data Lakes (Lake Formation)

While relational databases had dominated the landscape up until the mid 2000’s, it’s pretty easy to see that companies are now mixing and matching various platforms, and that relational databases will become less and less prevalent over time.

Future of SQL

So if relational databases are becoming less popular, what does that mean for the future of SQL? While many people think of SQL as the query language used for relational databases, the SQL language has taken on a life of its own in recent years, mostly because it is the used by millions of people (both programmers and non-programmers) and has been embedded into thousands of applications (such as just about every reporting and analytic engine). There have also been many attempts to graft the SQL language onto non-relational databases over the years, such as Hive for Hadoop. One tool that I find particularly interesting is Apache Drill, which is a SQL engine which can query many kinds of data, stored either locally or on just about any distributed file system. In a later chapter, I will demonstrate how to use Apache Drill to query the Sakila sample data set, both in MySQL and in MongoDB.

Chapter 18. SQL and Big Data

While most of the content in this book covers the various features of the SQL language when using a relational database such as MySQL, the data landscape has changed quite a bit over the past decade, and SQL is changing to meet the needs of today's rapidly evolving environments. Many organizations which had used relational databases exclusively just a few years ago are now also housing data in Hadoop clusters, data lakes, and NoSQL databases. At the same time, companies are struggling to find ways to gain insights from the ever-growing volumes of data, and the fact that this data is now spread across multiple data stores, perhaps both onsite and in the cloud, makes this a daunting task.

Because SQL is used by millions of people and has been integrated into thousands of applications, it makes sense to leverage SQL to harness this data and make it actionable. Before this can happen, however, the SQL language needs to evolve in order to work with semi-structured and unstructured data, and a new breed of tools has emerged to meet this challenge. This chapter will use one of these tools to demonstrate how data in different formats and stored on different servers can be brought together.

Apache Drill

There have been numerous tools and interfaces developed to allow SQL access to data stored in Hadoop, NoSQL, Spark, and cloud-based distributed file systems. Examples include Hive, which was one of the first attempts to allow users to query data stored in Hadoop, and Spark SQL, which is a library used to query data stored in various formats from within Spark. One relative newcomer is the open-source Apache Drill, which first hit the scene in 2015 and has some compelling features:

- facilitates queries across multiple data formats, including delimited data, JSON, Parquet, and log files
- connects to relational databases, Hadoop, NoSQL, HBase, Kafka
- allows creation of custom plug-ins to connect to most any other data store
- requires no up-front schema definitions
- supports the SQL:2003 standard
- works with popular BI tools like Tableau

Using Drill, you can connect to any number of data sources and begin querying, without the need to first set up a metadata repository. While it is beyond the scope of this book to discuss the installation and configuration options for Apache Drill, if you are interested in learning more I highly recommend “Learning Apache Drill” by Charles Givre and Paul Rogers (O’Reilly).

Drill and MySQL

Let’s start by running some Drill queries against the Sakila sample database used for the examples in this book. After loading the JDBC driver for MySQL and configuring Drill to connect to my local MySQL database, I should be able to run most of the example queries from earlier chapters. The first step is to choose a database:

```
apache drill (information_schema)> use mysql.sakila;
+------+------------------------------------------+
|  ok  |                 summary                  |
+------+------------------------------------------+
| true | Default schema changed to [mysql.sakila] |
+------+------------------------------------------+
1 row selected (0.062 seconds)
```

After choosing the database, Drill includes a simple command to show all of the tables available in the chosen schema:

```
apache drill (mysql.sakila)> show tables;
+--------------+----------------------------+
| TABLE_SCHEMA |         TABLE_NAME         |
+--------------+----------------------------+
| mysql.sakila | actor                      |
| mysql.sakila | address                    |
| mysql.sakila | category                   |
| mysql.sakila | city                       |
| mysql.sakila | country                    |
| mysql.sakila | customer                   |
| mysql.sakila | film                       |
| mysql.sakila | film_actor                 |
| mysql.sakila | film_category              |
| mysql.sakila | film_text                  |
| mysql.sakila | inventory                  |
| mysql.sakila | language                   |
| mysql.sakila | payment                    |
| mysql.sakila | rental                     |
| mysql.sakila | sales                      |
| mysql.sakila | staff                      |
| mysql.sakila | store                      |
| mysql.sakila | actor_info                 |
| mysql.sakila | customer_list              |
| mysql.sakila | film_list                  |
| mysql.sakila | nicer_but_slower_film_list |
```

```
| mysql.sakila | sales_by_film_category     |
| mysql.sakila | sales_by_store             |
| mysql.sakila | staff_list                 |
+--------------+----------------------------+
24 rows selected (0.147 seconds)
```

Everything looks good, so it's time to run some queries. Here's a simple 2-table join from the Joins chapter:

```
apache drill (mysql.sakila)> SELECT a.address_id, a.address,
ct.city
. . . . . . . . . . . . . )> FROM address a
. . . . . . . . . . . . . )>   INNER JOIN city ct
. . . . . . . . . . . . . )>   ON a.city_id = ct.city_id
. . . . . . . . . . . . . )> WHERE a.district =
'California';
+------------+------------------------+----------------+
| address_id |        address         |      city      |
+------------+------------------------+----------------+
| 6          | 1121 Loja Avenue       | San Bernardino |
| 18         | 770 Bydgoszcz Avenue   | Citrus Heights |
| 55         | 1135 Izumisano Parkway | Fontana        |
| 116        | 793 Cam Ranh Avenue    | Lancaster      |
| 186        | 533 al-Ayn Boulevard   | Compton        |
| 218        | 226 Brest Manor        | Sunnyvale      |
| 274        | 920 Kumbakonam Loop    | Salinas        |
| 425        | 1866 al-Qatif Avenue   | El Monte       |
| 599        | 1895 Zhezqazghan Drive | Garden Grove   |
+------------+------------------------+----------------+
9 rows selected (3.523 seconds)
```

The next query comes from the Grouping and Aggregates chapter and includes both a Group By and a Having clause:

```
apache drill (mysql.sakila)> SELECT fa.actor_id, f.rating,
. . . . . . . . . . . . . )>   count(*) num_films
. . . . . . . . . . . . . )> FROM film_actor fa
. . . . . . . . . . . . . )>   INNER JOIN film f
. . . . . . . . . . . . . )>   ON fa.film_id = f.film_id
. . . . . . . . . . . . . )> WHERE f.rating IN ('G','PG')
. . . . . . . . . . . . . )> GROUP BY fa.actor_id, f.rating
. . . . . . . . . . . . . )> HAVING count(*) > 9;
+----------+--------+-----------+
```

```
| actor_id | rating | num_films |
+----------+--------+-----------+
| 137      | PG     | 10        |
| 37       | PG     | 12        |
| 180      | PG     | 12        |
| 7        | G      | 10        |
| 83       | G      | 14        |
| 129      | G      | 12        |
| 111      | PG     | 15        |
| 44       | PG     | 12        |
| 26       | PG     | 11        |
| 92       | PG     | 12        |
| 17       | G      | 12        |
| 158      | PG     | 10        |
| 147      | PG     | 10        |
| 14       | G      | 10        |
| 102      | PG     | 11        |
| 133      | PG     | 10        |
+----------+--------+-----------+
16 rows selected (0.277 seconds)
```

Finally, here's a query from the Analytic Functions chapter which includes 3 different ranking functions:

```
apache drill (mysql.sakila)> SELECT customer_id, count(*)
num_rentals,
. . . . . . . . . . . . . . )>   row_number()
. . . . . . . . . . . . . . )>     over (order by count(*)
desc)
. . . . . . . . . . . . . . )>       row_number_rnk,
. . . . . . . . . . . . . . )>   rank()
. . . . . . . . . . . . . . )>     over (order by count(*)
desc) rank_rnk,
. . . . . . . . . . . . . . )>   dense_rank()
. . . . . . . . . . . . . . )>     over (order by count(*)
desc)
. . . . . . . . . . . . . . )>       dense_rank_rnk
. . . . . . . . . . . . . . )> FROM rental
. . . . . . . . . . . . . . )> GROUP BY customer_id
. . . . . . . . . . . . . . )> ORDER BY 2 desc;
+-------------+-------------+----------------+----------+---
-------------+
| customer_id | num_rentals | row_number_rnk | rank_rnk |
dense_rank_rnk |
+-------------+-------------+----------------+----------+---
```

```
-------------+
| 148         | 46          | 1              | 1         |
1             |
| 526         | 45          | 2              | 2         |
2             |
| 144         | 42          | 3              | 3         |
3             |
| 236         | 42          | 4              | 3         |
3             |
| 75          | 41          | 5              | 5         |
4             |
| 197         | 40          | 6              | 6         |
5             |
…
| 248         | 15          | 595            | 594       |
30            |
| 61          | 14          | 596            | 596       |
31            |
| 110         | 14          | 597            | 596       |
31            |
| 281         | 14          | 598            | 596       |
31            |
| 318         | 12          | 599            | 599       |
32            |
+-------------+-------------+----------------+----------+---
-------------+
599 rows selected (1.827 seconds)
```

It looks like Drill does a pretty good job querying MySQL, but you will need to keep in mind that Drill works with many relational databases, not just MySQL, so some features of the language may differ (e.g. data conversion functions). For more information, you can read about Drill's SQL implementation at http://drill.apache.org/docs/sql-reference/.

Drill and MongoDB

After using Drill to query the sample Sakila data in MySQL, I felt that the next logical step would be to convert the Sakila data to another commonly-used format, store it in a non-relational database, and use Drill to query the data. I decided to convert the data to JSON (JavaScript Object Notation) and store it in MongoDB, which is one of the more popular NoSQL platforms for document storage. Fortunately, I discovered that fellow author Guy Harrison[1] had the same idea a couple of years ago, and he was kind enough to share his files with me. Drill includes a plug-in for MongoDB, so it was relatively easy to load Guy's JSON files into Mongo and begin writing queries.

Before diving into the queries, let's take a look at the structure of the JSON files, since it isn't in normalized form. The first of the two JSON files is films.json:

```
{"_id":1,
 "Actors":[
   {"First name":"PENELOPE","Last
name":"GUINESS","actorId":1},
   {"First name":"CHRISTIAN","Last
name":"GABLE","actorId":10},
   {"First name":"LUCILLE","Last
name":"TRACY","actorId":20},
   {"First name":"SANDRA","Last name":"PECK","actorId":30},
   {"First name":"JOHNNY","Last name":"CAGE","actorId":40},
   {"First name":"MENA","Last name":"TEMPLE","actorId":53},
   {"First name":"WARREN","Last
name":"NOLTE","actorId":108},
   {"First name":"OPRAH","Last
name":"KILMER","actorId":162},
   {"First name":"ROCK","Last
name":"DUKAKIS","actorId":188},
   {"First name":"MARY","Last
name":"KEITEL","actorId":198}],
 "Category":"Documentary",
 "Description":"A Epic Drama of a Feminist And a Mad
Scientist
    who must Battle a Teacher in The Canadian Rockies",
 "Length":"86",
 "Rating":"PG",
```

```
 "Rental Duration":"6",
 "Replacement Cost":"20.99",
 "Special Features":"Deleted Scenes,Behind the Scenes",
 "Title":"ACADEMY DINOSAUR"},
{"_id":2,
 "Actors":[
   {"First name":"BOB","Last name":"FAWCETT","actorId":19},
   {"First name":"MINNIE","Last
name":"ZELLWEGER","actorId":85},
   {"First name":"SEAN","Last name":"GUINESS","actorId":90},
   {"First name":"CHRIS","Last name":"DEPP","actorId":160}],
 "Category":"Horror",
 "Description":"A Astounding Epistle of a Database
Administrator
    And a Explorer who must Find a Car in Ancient China",
 "Length":"48",
 "Rating":"G",
 "Rental Duration":"3",
 "Replacement Cost":"12.99",
 "Special Features":"Trailers,Deleted Scenes",
 "Title":"ACE GOLDFINGER"},
...
{"_id":999,
 "Actors":[
   {"First name":"CARMEN","Last name":"HUNT","actorId":52},
   {"First name":"MARY","Last name":"TANDY","actorId":66},
   {"First name":"PENELOPE","Last
name":"CRONYN","actorId":104},
   {"First name":"WHOOPI","Last name":"HURT","actorId":140},
   {"First name":"JADA","Last name":"RYDER","actorId":142}],
 "Category":"Children",
 "Description":"A Fateful Reflection of a Waitress And a
Boat
    who must Discover a Sumo Wrestler in Ancient China",
 "Length":"101",
 "Rating":"R",
 "Rental Duration":"5",
 "Replacement Cost":"28.99",
 "Special Features":"Trailers,Deleted Scenes",
 "Title":"ZOOLANDER FICTION"}
{"_id":1000,
 "Actors":[
   {"First name":"IAN","Last name":"TANDY","actorId":155},
   {"First name":"NICK","Last
name":"DEGENERES","actorId":166},
   {"First name":"LISA","Last
name":"MONROE","actorId":178}],
 "Category":"Comedy",
```

```
 "Description":"A Intrepid Panorama of a Mad Scientist And a
Boy
    who must Redeem a Boy in A Monastery",
 "Length":"50",
 "Rating":"NC-17",
 "Rental Duration":"3",
 "Replacement Cost":"18.99",
 "Special Features":
 "Trailers,Commentaries,Behind the Scenes",
 "Title":"ZORRO ARK"}
```

There are 1,000 documents in this collection, and each document contains a number of scalar attributes (Title, Rating, _id) but also includes an array called Actors, which contains 1 to N elements consisting of the actorId, First Name, and Last Name attributes for every actor appearing in the film. Therefore, this file contains all of the data found in the Actor, Film, and Film_Actor tables within the MySQL Sakila database.

The second file is customer.json, which combines data from the Customer, Address, City, Country, Rental, and Payment tables from the MySQL Sakila database:

```
{"_id":1,
 "Address":"1913 Hanoi Way",
 "City":"Sasebo",
 "Country":"Japan",
 "District":"Nagasaki",
 "First Name":"MARY",
 "Last Name":"SMITH",
 "Phone":"28303384290",
 "Rentals":[
   {"rentalId":1185,
    "filmId":611,
    "staffId":2,
    "Film Title":"MUSKETEERS WAIT",
    "Payments":[
      {"Payment Id":3,"Amount":5.99,"Payment Date":"2005-06-
15 00:54:12"}],
    "Rental Date":"2005-06-15 00:54:12.0",
```

```
    "Return Date":"2005-06-23 02:42:12.0"},
   {"rentalId":1476,
    "filmId":308,
    "staffId":1,
    "Film Title":"FERRIS MOTHER",
    "Payments":[
      {"Payment Id":5,"Amount":9.99,"Payment Date":"2005-06-
15 21:08:46"}],
    "Rental Date":"2005-06-15 21:08:46.0",
    "Return Date":"2005-06-25 02:26:46.0"},
...
   {"rentalId":14825,
    "filmId":317,
    "staffId":2,
    "Film Title":"FIREBALL PHILADELPHIA",
    "Payments":[
      {"Payment Id":30,"Amount":1.99,"Payment Date":"2005-
08-22 01:27:57"}],
    "Rental Date":"2005-08-22 01:27:57.0",
    "Return Date":"2005-08-27 07:01:57.0"}
  ]
}
```

This file contains 599 entries (only 1 is shown above), which is loaded into Mongo as 599 documents in the **customers** collection. Each document contains the information about a single customer, along with all of the rentals and associated payments made by that customer. Furthermore, the documents contain nested arrays, since each rental in the **Rentals** array also contains an array of **Payments**.

After the JSON files have been loaded, the Mongo database contains two collections (**films** and **customers**), and the data in these collections spans 9 different tables from the MySQL Sakila database. This is a fairly typical scenario, since application programmers typically work with collections, and generally prefer not to deconstruct their data for storage into normalized relational tables.

The challenge from an SQL perspective is to determine how to flatten this data so that it behaves as if it were stored in multiple tables.

To illustrate, let's construct the following query against the **films** collection: find all actors who have appeared in 10 or more films rated either G or PG. Here's what the raw data looks like:

```
apache drill (mongo.sakila)> SELECT Rating, Actors
. . . . . . . . . . . . . . )> FROM films
. . . . . . . . . . . . . . )> WHERE Rating IN ('G','PG');
+--------+------------------------------------------------------------
--------------+
| Rating |
Actors                    |
+--------+------------------------------------------------------------
--------------+
| PG     |[{"First name":"PENELOPE","Last
name":"GUINESS","actorId":"1"},
           {"First name":"FRANCES","Last name":"DAY-
LEWIS","actorId":"48"},
           {"First name":"ANNE","Last
name":"CRONYN","actorId":"49"},
           {"First name":"RAY","Last
name":"JOHANSSON","actorId":"64"},
           {"First name":"PENELOPE","Last
name":"CRONYN","actorId":"104"},
           {"First name":"HARRISON","Last
name":"BALE","actorId":"115"},
           {"First name":"JEFF","Last
name":"SILVERSTONE","actorId":"180"},
           {"First name":"ROCK","Last
name":"DUKAKIS","actorId":"188"}] |
| PG     |[{"First name":"UMA","Last
name":"WOOD","actorId":"13"},
           {"First name":"HELEN","Last
name":"VOIGHT","actorId":"17"},
           {"First name":"CAMERON","Last
name":"STREEP","actorId":"24"},
           {"First name":"CARMEN","Last
name":"HUNT","actorId":"52"},
           {"First name":"JANE","Last
name":"JACKMAN","actorId":"131"},
           {"First name":"BELA","Last
```

```
name":"WALKEN","actorId":"196"}] |
...
| G      |[{"First name":"ED","Last
name":"CHASE","actorId":"3"},
           {"First name":"JULIA","Last
name":"MCQUEEN","actorId":"27"},
           {"First name":"JAMES","Last
name":"PITT","actorId":"84"},
           {"First name":"CHRISTOPHER","Last
name":"WEST","actorId":"163"},
           {"First name":"MENA","Last
name":"HOPPER","actorId":"170"}] |
+--------+------------------------------------------------------------
--------------+
372 rows selected (0.432 seconds)
```

The Actors field is an array of one or more Actor documents. In order to interact with this data as if it were a table, the **flatten** command can be used to turn the array into a nested table containing 3 fields:

```
apache drill (mongo.sakila)> SELECT f.Rating,
flatten(Actors) actor_list
. . . . . . . . . . . . . . )>   FROM films f
. . . . . . . . . . . . . . )>   WHERE f.Rating IN ('G','PG');
+--------+------------------------------------------------------------
--------------+
| Rating |
actor_list                         |
+--------+------------------------------------------------------------
--------------+
| PG     | {"First name":"PENELOPE","Last
name":"GUINESS","actorId":"1"}  |
| PG     | {"First name":"FRANCES","Last name":"DAY-
LEWIS","actorId":"48"}|
| PG     | {"First name":"ANNE","Last
name":"CRONYN","actorId":"49"}      |
| PG     | {"First name":"RAY","Last
name":"JOHANSSON","actorId":"64"}    |
| PG     | {"First name":"PENELOPE","Last
name":"CRONYN","actorId":"104"} |
| PG     | {"First name":"HARRISON","Last
name":"BALE","actorId":"115"}   |
| PG     | {"First name":"JEFF","Last
name":"SILVERSTONE","actorId":"180"}|
| PG     | {"First name":"ROCK","Last
```

```
name":"DUKAKIS","actorId":"188"}    |
| PG     | {"First name":"UMA","Last
name":"WOOD","actorId":"13"}         |
| PG     | {"First name":"HELEN","Last
name":"VOIGHT","actorId":"17"}     |
| PG     | {"First name":"CAMERON","Last
name":"STREEP","actorId":"24"}   |
| PG     | {"First name":"CARMEN","Last
name":"HUNT","actorId":"52"}      |
| PG     | {"First name":"JANE","Last
name":"JACKMAN","actorId":"131"}    |
| PG     | {"First name":"BELA","Last
name":"WALKEN","actorId":"196"}     |
...
| G      | {"First name":"ED","Last
name":"CHASE","actorId":"3"}          |
| G      | {"First name":"JULIA","Last
name":"MCQUEEN","actorId":"27"}    |
| G      | {"First name":"JAMES","Last
name":"PITT","actorId":"84"}       |
| G      | {"First name":"CHRISTOPHER","Last
name":"WEST","actorId":"163"}|
| G      | {"First name":"MENA","Last
name":"HOPPER","actorId":"170"}     |
+--------+-----------------------------------------------
--------------+
2,119 rows selected (0.718 seconds)      |
```

This query returns 2,119 rows, rather than the 372 rows returned by the previous query, which indicates that there are an average of 5.7 actors appearing in each G or PG film. This query can then be wrapped in a subquery and used to group the data by rating and actor, as in :

```
apache drill (mongo.sakila)> SELECT g_pg_films.Rating,
. . . . . . . . . . . . . . )>   g_pg_films.actor_list.`First
name` first_name,
. . . . . . . . . . . . . . )>   g_pg_films.actor_list.`Last
name` last_name,
. . . . . . . . . . . . . . )>   count(*) num_films
. . . . . . . . . . . . . . )> FROM
. . . . . . . . . . . . . . )>  (SELECT f.Rating,
flatten(Actors) actor_list
```

```
. . . . . . . . . . . . . . )>   FROM films f
. . . . . . . . . . . . . . )>   WHERE f.Rating IN ('G','PG')
. . . . . . . . . . . . . . )>  ) g_pg_films
. . . . . . . . . . . . . . )> GROUP BY g_pg_films.Rating,
. . . . . . . . . . . . . . )>   g_pg_films.actor_list.`First
name`,
. . . . . . . . . . . . . . )>   g_pg_films.actor_list.`Last
name`
. . . . . . . . . . . . . . )> HAVING count(*) > 9;
+--------+-----------+------------+----------+
| Rating | first_name |  last_name  | num_films |
+--------+-----------+------------+----------+
| PG     | JEFF       | SILVERSTONE | 12        |
| G      | GRACE      | MOSTEL      | 10        |
| PG     | WALTER     | TORN        | 11        |
| PG     | SUSAN      | DAVIS       | 10        |
| PG     | CAMERON    | ZELLWEGER   | 15        |
| PG     | RIP        | CRAWFORD    | 11        |
| PG     | RICHARD    | PENN        | 10        |
| G      | SUSAN      | DAVIS       | 13        |
| PG     | VAL        | BOLGER      | 12        |
| PG     | KIRSTEN    | AKROYD      | 12        |
| G      | VIVIEN     | BERGEN      | 10        |
| G      | BEN        | WILLIS      | 14        |
| G      | HELEN      | VOIGHT      | 12        |
| PG     | VIVIEN     | BASINGER    | 10        |
| PG     | NICK       | STALLONE    | 12        |
| G      | DARYL      | CRAWFORD    | 12        |
| PG     | MORGAN     | WILLIAMS    | 10        |
| PG     | FAY        | WINSLET     | 10        |
+--------+-----------+------------+----------+
18 rows selected (0.466 seconds)
```

The inner query uses the **flatten** command to create one row for every actor who has appeared in a G or PG movie, and the outer query simply performs a grouping on this data set.

Next, let's try to write a query against the **customers** collection in Mongo. This should prove a bit more challenging, since each document contains an array of film rentals, each of which contains an array of payments. To make it a little more interesting, let's also join to the **films** collection in order to see how Drill handles joins. The query should return all customers who have spent more then $80 to rent films rated either G or PG. Here's what it looks like:

```
apache drill (mongo.sakila)> SELECT first_name, last_name,
. . . . . . . . . . . . . . )>
sum(cast(cust_payments.payment_data.Amount
. . . . . . . . . . . . . . )>        as decimal(4,2)))
tot_payments
. . . . . . . . . . . . . . )> FROM
. . . . . . . . . . . . . . )>  (SELECT cust_data.first_name,
. . . . . . . . . . . . . . )>     cust_data.last_name,
. . . . . . . . . . . . . . )>     f.Rating,
. . . . . . . . . . . . . . )>
flatten(cust_data.rental_data.Payments)
. . . . . . . . . . . . . . )>       payment_data
. . . . . . . . . . . . . . )>   FROM films f
. . . . . . . . . . . . . . )>     INNER JOIN
. . . . . . . . . . . . . . )>    (SELECT c.`First Name`
first_name,
. . . . . . . . . . . . . . )>       c.`Last Name` last_name,
. . . . . . . . . . . . . . )>       flatten(c.Rentals)
rental_data
. . . . . . . . . . . . . . )>     FROM customers c
. . . . . . . . . . . . . . )>    ) cust_data
. . . . . . . . . . . . . . )>     ON f._id =
cust_data.rental_data.filmID
. . . . . . . . . . . . . . )>   WHERE f.Rating IN ('G','PG')
. . . . . . . . . . . . . . )>  ) cust_payments
. . . . . . . . . . . . . . )> GROUP BY first_name, last_name
. . . . . . . . . . . . . . )> HAVING
. . . . . . . . . . . . . . )>
sum(cast(cust_payments.payment_data.Amount
. . . . . . . . . . . . . . )>        as decimal(4,2))) > 80;
+------------+-----------+-------------+
| first_name | last_name | tot_payments |
+------------+-----------+-------------+
| ELEANOR    | HUNT      | 85.80        |
| GORDON     | ALLARD    | 85.86        |
| CLARA      | SHAW      | 86.83        |
| JACQUELINE | LONG      | 86.82        |
| KARL       | SEAL      | 89.83        |
| PRISCILLA  | LOWE      | 95.80        |
| MONICA     | HICKS     | 85.82        |
| LOUIS      | LEONE     | 95.82        |
```

```
| JUNE         | CARROLL    | 88.83          |
| ALICE        | STEWART    | 81.82          |
+------------+-----------+--------------+
10 rows selected (1.658 seconds)
```

The innermost query, which I named **cust_data**, flattens the **Rental** array so that the **cust_payments** query can join to the **films** collection and also flatten the **Payment** array. The outermost query groups the data by customer name and applies a **having** clause to filter out customers who spent $80 or less on films rated G or PG.

Drill with Multiple Data Sources

So far, I have used Drill to join multiple tables stored in the same database, but what if the data is stored in different databases? For example, let's say the customer/rental/payment data is stored in MongoDB, but the catalog of film/actor data is stored in MySQL. As long as Drill is configured to connect to both databases, you just need to describe where to find the data. Here's the query from the previous section, but instead of joining to the **films** collection stored in MongoDB the join specifies the **film** table stored in MySQL:

```
apache drill (mongo.sakila)> SELECT first_name, last_name,
. . . . . . . . . . . . . . )>   sum(cast(cust_payments.payme
. . . . . . . . . . . . . . )>         as decimal(4,2))) tot_
. . . . . . . . . . . . . . )> FROM
. . . . . . . . . . . . . . )>  (SELECT cust_data.first_name,
. . . . . . . . . . . . . . )>     cust_data.last_name,
. . . . . . . . . . . . . . )>     f.Rating,
. . . . . . . . . . . . . . )>     flatten(cust_data.rental_d
. . . . . . . . . . . . . . )>       payment_data
. . . . . . . . . . . . . . )>   FROM mysql.sakila.film f
. . . . . . . . . . . . . . )>     INNER JOIN
```

```
. . . . . . . . . . . . . . )>    (SELECT c.`First Name` firs
. . . . . . . . . . . . . . )>       c.`Last Name` last_name,
. . . . . . . . . . . . . . )>       flatten(c.Rentals) renta
. . . . . . . . . . . . . . )>     FROM mongo.sakila.customer
. . . . . . . . . . . . . . )>    ) cust_data
. . . . . . . . . . . . . . )>     ON f.film_id =
. . . . . . . . . . . . . . )>       cast(cust_data.rental_da
. . . . . . . . . . . . . . )>   WHERE f.rating IN ('G','PG')
. . . . . . . . . . . . . . )>  ) cust_payments
. . . . . . . . . . . . . . )> GROUP BY first_name, last_name
. . . . . . . . . . . . . . )> HAVING
. . . . . . . . . . . . . . )>   sum(cast(cust_payments.payme
. . . . . . . . . . . . . . )>         as decimal(4,2))) > 80
+------------+-----------+--------------+
| first_name | last_name | tot_payments |
+------------+-----------+--------------+
| LOUIS      | LEONE     | 95.82        |
| JACQUELINE | LONG      | 86.82        |
| CLARA      | SHAW      | 86.83        |
| ELEANOR    | HUNT      | 85.80        |
| JUNE       | CARROLL   | 88.83        |
| PRISCILLA  | LOWE      | 95.80        |
| ALICE      | STEWART   | 81.82        |
| MONICA     | HICKS     | 85.82        |
| GORDON     | ALLARD    | 85.86        |
| KARL       | SEAL      | 89.83        |
+------------+-----------+--------------+
10 rows selected (1.874 seconds)
```

Since I'm using multiple databases in the same query, I specified the full path to each table/collection to make it clear as to where the data is being sourced. This is where Drill really shines, since I can combine data from multiple sources in the same query without having to transform and load the data from one source to another.

Someday, we may look back on relational databases with a feeling of nostalgia, similar to how we remember things like floppy discs, company pensions, and civility. SQL will likely live on, however, and it will be tools like Apache Drill that help keep SQL relevant for years to come.

---

1 For those readers who would benefit from a high-level overview of the new and emerging database trends, I recommend Guy Harrison's "Next Generation Databases", published in 2015 by Apress.

About the Author

Alan Beaulieu has been designing, building, and implementing custom database applications for more than 25 years. He is the author of *Learning SQL*, Second Edition, and coauthored *Mastering Oracle SQL*, Second Edition (both O’Reilly), and has written an online course on SQL for the University of California. He currently runs his own consulting company that specializes in database design and development in the fields of financial services and telecommunications. Alan has a BS in operations research from the Cornell University School of Engineering. He lives in Massachusetts with his wife and two daughters.