5

Managing Databases, Tables, and Indexes

In the first four chapters of the book, you were provided with the information necessary to install and use MySQL and design relational databases that could be implemented in MySQL. You were also provided with a foundation in the principles of SQL and the relational model. From all this information, you should now have the background you need to begin creating databases and adding the objects that those databases should contain.

The first step in setting up a MySQL database is to create the actual database object, which serves as a container for the tables in that database. The database acts as a central point of administration for the tables in the database. The actual data is stored in the tables, which provide a structured organization for the data and maintain the integrity of that data. Associated with each table is a set of indexes that facilitate access to the data in the tables.

In this chapter, you learn how to create and manage the database object, the tables in the database, and the indexes associated with the tables. To provide you with the information necessary to perform all these tasks, the chapter covers the following topics:

- How to create a database and specify a character set and collation name for that database. You also learn how to modify database properties and remove a database from your system.
- The information necessary to create a table, define columns in that table, add constraints to the table, create indexes on the table, and specify the type of table to be created.
- Information about the various types of indexes that you can add to a table, how to add indexes to a table, and how to remove an index from a table.
- How to retrieve information about databases and tables so that you can see what databases and tables exist and how those databases and tables are configured.

Managing Databases

Once you install MySQL and are comfortable using the tools in MySQL — particularly the mysql client utility — you can begin creating databases in the MySQL environment. Recall from Chapter 4

that when you add a database to MySQL, a corresponding subdirectory is automatically added to the data directory. Any tables added to the database appear as files in the subdirectory. If you then remove a database, the corresponding subdirectory and files are also deleted.

As you can see, the first step in setting up a database in MySQL is to create the database object and, subsequently, the corresponding subdirectories. From there, you can modify the database definition or delete the database. This section discusses each of these tasks.

Creating Databases

Creating a database in MySQL is one of the easier tasks that you're likely to perform. At its most basic, a database definition statement requires only the keywords `CREATE DATABASE`, followed by the name of the new database, as shown in the following syntax:

```
<database definition>::=
CREATE DATABASE [IF NOT EXISTS] <database name>
[[DEFAULT] CHARACTER SET <character set name>]
[[DEFAULT] COLLATE <collation name>]
```

As the syntax shows, very few components of the `CREATE DATABASE` statement are actually required. The statement does include several optional elements. The first of these — `IF NOT EXISTS` — determines how MySQL responds to a `CREATE DATABASE` statement if a database with the same name already exists. If a database already exists and the `IF NOT EXISTS` clause is not used, MySQL returns an error. If the clause is used, MySQL returns a warning, without returning any errors. Regardless of whether the clause is included in the statement, the effect on the database is the same: If a database with the same name exists, no new database is created.

The next optional components of the `CREATE DATABASE` statement are the `CHARACTER SET` clause and the `COLLATE` clause. You can specify either one of the two clauses or both. (The `DEFAULT` keyword is optional in either case and has no effect on the outcome.) The `CHARACTER SET` clause specifies the default character set to use for a new database, and the `COLLATE` clause specifies which default collation to use. A *character set* is a collection of letters, numbers, and symbols that create the values in a database. For example, A, B, C, a, b, c, 1, 2, 3, >, +, and * are all part of a character set. A *collation* is a named sorting order used for a specified character set. Collations define the way in which values made up of a particular character set are compared, sorted, and grouped together. Most character sets have one or more collations associated with them. For example, some of the collations associated with the default character set, `latin1`, include `latin1_bin`, `latin1_general_ci`, and `latin1_swedish_ci`, which is the default collation. If you do not specify the `CHARACTER SET` or `COLLATION` clause, the database uses the default MySQL character set or collation.

> *You can view the character sets and collations available in your system by executing the* `SHOW CHARACTER SET` *and* `SHOW COLLATION` *statements in the mysql client utility. Also note that character sets and collations affect only string data (letters, numbers, and symbols), as opposed to all numerical data or data related to dates and times.*

Now that you have an understanding of the syntax used to create a database, take a look at a couple examples. The first example is a basic `CREATE DATABASE` statement that includes no optional components:

```
CREATE DATABASE BookSales;
```

When you execute the statement, a database named BookSales is added to your system. The database uses the default character set and collation because you specified neither.

> *When you create databases and tables in Windows, all names are converted to lowercase. Because Windows filenames and directory names are case insensitive, it follows that case is not a factor when specifying database and table names. In Linux and other Unix-like operating systems, the case is preserved. Because filenames and directory names are case sensitive, you must specify the correct case when referencing database and table names.*

In the next example, the `CREATE DATABASE` statement specifies the character set and collation:

```
CREATE DATABASE BookSales
DEFAULT CHARACTER SET latin1
DEFAULT COLLATE latin1_bin;
```

In this example, the `CHARACTER SET` clause specifies the `latin1` character set, and the `COLLATE` clause specifies the `latin1_bin` collation. In both cases, you use the `DEFAULT` keyword, but it isn't required. Executing the statement creates a database named BookSales, which uses the specified character set and collation.

In the Try It Out sections in Chapter 4, you created a data model for the DVDRentals database. In the Try It Out examples in this chapter, however, you create the database based on that model. Later in the chapter, you create the tables in the DVDRentals database.

Try It Out Creating the DVDRentals Database

To create the database, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
CREATE DATABASE DVDRentals;
```

   You should receive a message indicating that your statement executed successfully.

2. In order to create tables or execute any statements in the context of the new database, you should switch over to that new database by using the following command:

```
use DVDRentals
```

   You should receive a message indicating the database change.

How It Works

In this exercise, you used a `CREATE DATABASE` statement to create the DVDRentals database. This is the database for which you created a data model in Chapter 4. Because you didn't specify any character set or collation, the database uses the default values, which for a standard MySQL installation are the character set `latin1` and the collation `latin1_swedish_ci`. Once you create the DVDRentals database, you can begin adding the necessary tables.

Modifying Databases

There might be times when you want to change the character set or collation used for your database. To do this, you can use an ALTER DATABASE statement to specify the new settings. As you can see from the following syntax, the ALTER DATABASE statement is similar to the CREATE DATABASE statement:

```
ALTER DATABASE <database name>
[[DEFAULT] CHARACTER SET <character set name>]
[[DEFAULT] COLLATE <collation name>]
```

In this statement, you must specify the ALTER DATABASE keywords and the name of the database, along with the CHARACTER SET clause, the COLLATE clause, or both. For either clause, simply specify the name of the character set and collation. For example, to change the character set to latin1 for the BookSales database, use the following ALTER DATABASE statement:

```
ALTER DATABASE BookSales
CHARACTER SET latin1;
```

As you can see, the statement specifies only a CHARACTER SET clause, which means the current collation remains the same.

Use caution when changing the character set for a database. In some cases, changing the character set can result in the database no longer supporting all the characters stored as data in the database.

Deleting Databases

Deleting a database from your system is merely a matter of executing a DROP DATABASE statement. The following syntax shows the components that make up the statement:

```
DROP DATABASE [IF EXISTS] <database name>
```

The statement requires only the DROP DATABASE keywords and the name of the database. In addition, you can specify the IF EXISTS clause. If you specify this clause and a database with that name doesn't exist, you receive a warning message rather than an error. Now take a look at an example of the DROP DATABASE statement:

```
DROP DATABASE BookSales;
```

This example removes the BookSales database from the system. When you remove a database, you also remove the tables in that database and any data contained in the table. As a result, you want to be extremely cautious whenever you execute the DROP DATABASE command.

Managing Tables

The next step in setting up a database, after creating the actual database object, is to add the necessary tables to that database. The tables provide a structure for storing and securing the data. All data exists within the structure of the tables, and all tables exist within the structure of the database. In addition to creating tables, you can also modify the table definitions or delete the tables from the database. This section explains how to perform each of these tasks.

Creating Tables

To create a table in MySQL, you must use the `CREATE TABLE` statement to define the columns in the table and configure the appropriate constraints on the table. The `CREATE TABLE` statement is one of the most complex SQL statements in MySQL. It contains numerous components and provides many options for defining the exact nature of a particular table. The following syntax represents the elements that make up a `CREATE TABLE` statement:

```
<table definition>::=
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
(<table element> [{, <table element>}...])
[<table option> [<table option>...]]

<table element>::=
<column definition>
| {[CONSTRAINT <constraint name>] PRIMARY KEY
     (<column name> [{, <column name>}...])}
| {[CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
     (<column name> [{, <column name>}...]) <reference definition>}
| {[CONSTRAINT <constraint name>] UNIQUE [INDEX] [<index name>]
     (<column name> [{, <column name>}...])}
| {{INDEX | KEY} [<index name>] (<column name> [{, <column name>}...])}
| {FULLTEXT [INDEX] [<index name>] (<column name> [{, <column name>}...])}

<column definition>::=
<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]
[PRIMARY KEY] [COMMENT '<string>'] [<reference definition>]

<type>::=
<numeric data type>
| <string data type>
| <data/time data type>

<reference definition>::=
REFERENCES <table name> [(<column name> [{, <column name>}...])]
[ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
[ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
[MATCH FULL | MATCH PARTIAL]

<table option>::=
{ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MYISAM}}
| <additional table options>
```

As you can see, many elements make up a `CREATE TABLE` statement. In fact, the syntax shown here is not the `CREATE TABLE` statement in its entirety. As you move through the chapter, other elements are introduced and some elements, which are beyond the scope of this book, are not discussed at all. Still, this chapter covers all the essential components, so by the end of the chapter, you should have a fairly comprehensive foundation on how to create a table definition.

Now take a closer look at the `CREATE TABLE` syntax. The best place to begin is at the first section:

```
<table definition>::=
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
(<table element> [{, <table element>}...])
[<table option> [<table option>...]]
```

This section represents the entire CREATE TABLE statement, with individual components being explained later in the syntax. The first line of the actual statement requires only the keywords CREATE TABLE, followed by the name of the new table. This line also contains two optional components. The first — TEMPORARY — indicates that this is a temporary table used only during the current session by the current user. A temporary table exists only as long as the session is open or the table is explicitly dropped. The second optional element is the IF NOT EXISTS clause. You've seen this clause before in the CREATE DATABASE statement. When you include it in your CREATE TABLE statement, a warning is generated, rather than an error, if a table by the same name already exists when you execute this statement.

The next line of syntax allows you to define the individual table elements, as represented by the <table element> placeholder. A table element is an individual object that is defined on a table, such as a column or PRIMARY KEY constraint. Each CREATE TABLE statement includes one or more table elements. If more than one table element exists, they are separated by commas. Regardless of how many table elements exist, they are all enclosed in a set of parentheses.

The last line in the first section of syntax allows you to define individual table options. Table options are options that apply to the table as a whole. For example, you can define the type of table that you want to create. All table options are optional; however, you can define as many as are necessary.

As you can see from the first section of syntax, a CREATE TABLE statement can be as simple or as complicated as you need to make it. The only required elements are the CREATE TABLE clause and at least one table element enclosed in parentheses, as shown in the following syntax:

```
CREATE TABLE <table name> (<table element>)
```

Because a table element is a required component, take a look at the next section of syntax:

```
<table element>::=
<column definition>
| {[CONSTRAINT <constraint name>] PRIMARY KEY
     (<column name> [{, <column name>}...])}
| {[CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
     (<column name> [{, <column name>}...]) <reference definition>}
| {[CONSTRAINT <constraint name>] UNIQUE [INDEX] [<index name>]
     (<column name> [{, <column name>}...])}
| {{INDEX | KEY} [<index name>] (<column name> [{, <column name>}...])}
| {FULLTEXT [INDEX] [<index name>] (<column name> [{, <column name>}...])}
```

A table element can represent one of many different options. The most commonly used option is the one represented by the <column definition> placeholder, which, as the name suggests, allows you to define a column to include in your table definition. You are likely, though, to use the other options with regularity. For this reason, the following sections examine each of these options individually.

Creating Column Definitions

A column definition is one type of table element that you can define in a table definition. You must create a column definition for each column that you want to include in your table. The following syntax provides you with the structure that you should use when creating a column definition:

```
<column definition>::=
<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]
[PRIMARY KEY] [COMMENT '<string>'] [<reference definition>]
```

As you can see, only two elements are required in a column definition: the column name (represented by the <column name> placeholder) and the data type (represented by the <type> placeholder). The name can be any acceptable identifier in MySQL, and the database can be any of the supported data types. Each additional element of the column definition is optional and, along with data types, they are discussed individually in the following sections.

Defining Data Types

As you recall from earlier chapters, a data type is a constraint placed on a particular column to limit the type of values that you can store in that column. MySQL supports three categories of data types, as represented by the following syntax:

```
<type>::=
<numeric data type>
| <string data type>
| <data/time data type>
```

Whenever you add a column to your table, you must define the column with a data type that is in one of these three categories. Each category of types has its own specific characteristics, and each places restrictions on the type of data that you can include in a particular column. Take a closer look at each category to better understand the characteristics of each.

Numeric Data Types

As the name suggests, *numeric* data types are concerned with numbers. If you have a column that will contain nothing but numbers, chances are that you want to configure that column with a numeric data type.

You can divide numeric data types into two categories, as the following syntax suggests:

```
<numeric data type>::=
<integer data type> [(<length>)] [UNSIGNED] [ZEROFILL]
| <fractional data type> [(<length>, <decimals>)] [UNSIGNED] [ZEROFILL]
```

Each of the two categories of integer data types supports several options. The first of these is represented by the <length> placeholder, which indicates the maximum number of displayed characters for a particular column. You can specify that the length be anywhere from 1 to 255. The fractional data types also include the <decimals> placeholder. This value indicates the number of decimal places to be displayed for a value in the column. You can specify that the number of decimal places be anywhere from 0 to 30; however, <decimals> must always be at least two less than <length>.

The next option available for the numeric data types is UNSIGNED. When this option follows a numeric data type, no negative values are permitted in the column. If you specify ZEROFILL, zeros are added to the beginning of a value so that the value is displayed with the number of characters represented by the <length> placeholder. For example, if you define <length> as 4 and you specify ZEROFILL, the number 53 displays as 0053. ZEROFILL is most useful when used in conjunction with a specified length. In addition, UNSIGNED is assumed when you specify ZEROFILL, even if UNSIGNED isn't explicitly specified. In other words, you can't use ZEROFILL for negative numbers.

Now take a look at the two categories of numeric data types. The first of these is the *integer* data type. Integer data types allow you to store only whole numbers in your column (no fractions or decimals). MySQL supports the integer data types shown in the following syntax:

```
<integer data type>::=
TINYINT | SMALLINT | MEDIUMINT | INT | INTEGER | BIGINT
```

Each of these data types specifies not only that whole numbers must be stored in the column, but also that the numbers stored must fall within a specific range of values, as shown in the following table.

Data type	Acceptable values	Storage requirements
TINYINT	Signed: –128 to 127 Unsigned: 0 to 255	1 byte
SMALLINT	Signed: –32768 to 32767 Unsigned: 0 to 65535	2 bytes
MEDIUMINT	Signed: –8388608 to 8388607 Unsigned: 0 to 16777215	3 bytes
INT	Signed: –2147483648 to 2147483647 Unsigned: 0 to 4294967295	4 bytes
INTEGER	Same values as the INT data type. (INTEGER is a synonym for INT.)	4 bytes
BIGINT	Signed: –9223372036854775808 to 9223372036854775807 Unsigned: 0 to 18446744073709551615	8 bytes

The range of acceptable values for each integer data type has nothing to do with the `<length>` placeholder. Whether you were to specify a length of 2 or 20, the stored value would be the same, as would be the value limitations. As the table demonstrates, signed values are different from unsigned values. If a column doesn't require negative values, using the `UNSIGNED` option increases the upper end of the range of stored values, although the storage requirements remain the same. For example, a value in a `TINYINT` column requires 1 byte of storage whether the column is signed or unsigned.

Now take a look at an example of a `CREATE TABLE` statement that includes two column definitions, one that uses the `SMALLINT` data type and one that uses the `INT` data type. The following statement creates a table named Inventory:

```
CREATE TABLE Inventory
(
   ProductID SMALLINT(4) UNSIGNED ZEROFILL,
   Quantity INT UNSIGNED
);
```

As you can see, the table includes a column named ProductID and a column named Quantity. The ProductID column is configured with a `SMALLINT` data type that specifies 4 as its maximum display length. Because a `SMALLINT` value can have a range of 0 to 65535, the display of 4 might not apply to all values. You would specify a display size smaller than the capacity only if you're certain that all digits will fall into that range. If your value does require the full five digits, they will all be displayed, despite the `<length>` value of 4. As a result, the only time including the length is useful is when you're also including the `ZEROFILL` option, which is the case in this column definition. As a result, no negative values are allowed and zeros are added to all values less than four characters wide.

The next column defined in this statement is the Quantity column. This column is defined with an `INT` data type, and it is also unsigned. As a result, negative numbers are not allowed in this column, which means that acceptable values can fall in the range of 0 to 4294967295.

Another thing to notice about this `CREATE TABLE` statement is that the column definitions are enclosed in parentheses and separated by a comma. All table elements, including column definitions, are treated in this manner.

Now take a look at the fractional data types, which are shown in the following syntax:

```
<fractional data type>::=
FLOAT | DOUBLE | DOUBLE PRECISION | REAL | DECIMAL | DEC | NUMERIC |FIXED
```

The *fractional* data types, unlike the integer data types, support the use of decimals. In fact, that is the key characteristic of these data types, which are described in the following table:

Data type	Description
FLOAT	An approximate numeric data type that uses 4 bytes of storage. The data type supports the following values: –3.402823466E+38 to –1.175494351E–38 0 1.175494351E–38 to 3.402823466E+38
DOUBLE	An approximate numeric data type that uses 8 bytes of storage. The data type supports the following values: –1.7976931348623157E+308 to –2.2250738585072014E–308 0 2.2250738585072014E–308 to 1.7976931348623157E+308
DOUBLE PRECISION	Synonym for the DOUBLE data type
REAL	Synonym for the DOUBLE data type
DECIMAL	An exact numeric data type whose range storage requirements depend on the <length> and <decimals> values specified in the column definition
DEC	Synonym for the DECIMAL data type
NUMERIC	Synonym for the DECIMAL data type
FIXED	Synonym for the DECIMAL data type

As described in the table, there are only three basic fractional data types: `FLOAT`, `DOUBLE`, and `DECIMAL`. The rest are synonyms for the `DOUBLE` and `DECIMAL` data types.

The `DOUBLE` data type supports a greater degree of precision than does the `FLOAT` data type. In other words, `DOUBLE` supports more decimal places than `FLOAT`. If you need the greater precision, you would use `DOUBLE`, although you should be aware that this doubles your storage requirements. In both cases,

values are stored as numeric data and are subject to errors caused by numbers being rounded, which is why they're referred to as *approximate* numeric types. Generally numbers are rounded according to the column <length> and <decimals> specifications, which can sometimes result in imprecise results.

The DECIMAL data type, which is referred to as an *exact* numeric data type, gets around the issue of round-off errors by storing the values as strings, with the <length> and <decimals> specifications determining storage requirements and range. You should use the DECIMAL data type when you require values to be completely accurate, such as when you're storing information about money. The drawback to using the DECIMAL data type is that there are trade-offs in performance compared to the approximate numeric types. For this reason, if you plan to store values that don't require the accuracy of the DECIMAL type, use FLOAT or DOUBLE.

Now take a look at a table definition that uses the DECIMAL and FLOAT data types. The following CREATE TABLE statement creates a table named Catalog:

```
CREATE TABLE Catalog
(
   ProductID SMALLINT,
   Price DECIMAL(7,2),
   Weight FLOAT(8,4)
);
```

As you can see, the table includes a DECIMAL column (named Price) and a FLOAT column (named Weight). The Price column contains a <length> value of 7 and a <decimals> value of 2. As a result, the values display with up to 7 characters and 2 decimal places, such as 46264.45 and 242.90.

> *Because of the storage requirements for* DECIMAL *values, positive* DECIMAL *values (as opposed to negative values) receive one extra character to display values. For example, a* DECIMAL *column that has a* <length> *value of 7 and a* <decimals> *value of 2 can actually have up to eight numeric characters — plus one character for the decimal point — for positive numbers, but only seven characters — plus one character for the negative sign and one for the decimal point — for negative numbers.* FLOAT *and* DOUBLE *values do not operate in the same way.*

The second column in the preceding example is configured with the FLOAT data type. In this case, the total display can be eight characters long, with four characters to the right of the decimal point. The implication in this case is that the Weight value does not have to be as exact as the Price value, so you don't have to worry about errors caused by values that have been rounded.

String Data Types

The string data types provide a great deal of flexibility for storing various types of data from individual bits to large files. String data types are normally used to store names and titles and any value that can include letters and numbers. MySQL supports four categories of string data types, as shown in the following syntax:

```
<string data type>::=
<character data type>
| <binary data type>
| <text data type>
| <list data type>
```

The character data types are the ones that you probably use the most often. As the following syntax shows, there are two types of character data types:

```
<character data type>::=
CHAR (<length>) [BINARY | ASCII | UNICODE]
VARCHAR (<length>) [BINARY]
```

The `CHAR` data type is a fixed-length character data type that can store up to 255 characters. The `<length>` placeholder specifies the number of characters stored. Although the actual value can be made up of fewer characters than the amount specified, the actual storage space is fixed at the specified amount. Take a look at an example to demonstrate how this works. The following table definition includes a column definition for the Category column:

```
CREATE TABLE Catalog
(
   ProductID SMALLINT,
   Description VARCHAR(40),
   Category CHAR(3),
   Price DECIMAL(7,2)
);
```

The Category column is defined with the `CHAR(3)` data type. As a result, the column can store zero to three characters per value, but the storage amount allotted to that value is always three bytes, one for each character.

The `CHAR` data type is an appropriate data type to use when you know how many characters most values in a column will consist of and when the values are made up of alphanumeric characters, as opposed to all numerals. If you don't know how many characters each value will be, you should use a `VARCHAR` data type. The `VARCHAR` data type also allows you to specify a maximum length; however, storage requirements are based on the actual number of characters, rather than on the `<length>` value.

Return to the preceding example. Notice that the Description column is configured with the `VARCHAR(40)` data type. This means that the values can be of varying length, up to 40 characters long. As a result, the amount of actual storage ranges between zero bytes and 40 bytes, depending on the actual value. For example, a value of "Bookcase" requires fewer bytes than a value of "Three-piece living room set."

The `VARCHAR` data type, like the `CHAR` data type, can store up to 255 characters. Along with the flexibility offered by `VARCHAR`, compared to `CHAR`, comes a performance cost. `CHAR` columns are processed more efficiently than `VARCHAR` columns, yet `CHAR` columns can result in wasted storage. Generally, for columns with values that vary widely in length, the `VARCHAR` data type might often be your best choice.

If you return again to the `<character data type>` syntax, you'll see that the `CHAR` data type allows you to specify the `BINARY`, `ASCII`, or `UNICODE` attribute, and the `VARCHAR` data type allows you to specify the `BINARY` attribute. The three attributes result in the following effects:

- **BINARY:** Makes sorting and comparisons case sensitive.
- **ASCII:** Assigns the `latin1` character set to the column.
- **UNICODE:** Assigns the `ucs2` character set to the column.

In addition to string data types, MySQL supports four types of binary data types, as shown in the following syntax:

```
<binary data type>::=
TINYBLOB | BLOB | MEDIUMBLOB | LONGBLOB
```

Binary data types support the storage of large amounts of data, such as image and sound files. These types are useful when you expect values to grow large or to vary widely. The four binary data types are identical except for the maximum amount of data that each one supports. The following table shows the maximum size of values permitted in a column configured with one of these data types.

Data type	Maximum size
TINYBLOB/TINYTEXT	255 characters (355 bytes)
BLOB/TEXT	65,535 characters (64 KB)
MEDIUMBLOB/MEDIUMTEXT	16,777,215 characters (16 MB)
LONGBLOB/LONGTEXT	4,294,967,295 characters (4 GB)

The text data types are also included in the table because they are similar to the binary data types and because the maximum size limitations are the same. The text data types are discussed later in this section.

The binary data types do not take any arguments. As with data in a VARCHAR column, the storage used for binary data varies according to the size of the value, but you do not specify a maximum length. When defining a column with a binary data type, you simply type the name of the data type in the column definition. For example, the following table definition includes a `BLOB` column named Photo:

```
CREATE TABLE Inventory
(
   ProductID SMALLINT UNSIGNED,
   Name VARCHAR(40),
   Photo BLOB,
   Quantity INT UNSIGNED
);
```

The Photo column can store binary data up to 64 KB in size. The assumption in this case is that a photo can be taken of the product and saved in a small enough file to fit into this column. If you anticipate that the photos might be larger, you should step this up to `MEDIUMBLOB`.

The text data types are very similar to the binary data types and, as the following syntax shows, have a direct counterpart to each of the four binary data types:

```
<text data type>::=
TINYTEXT | TEXT | MEDIUMTEXT | LONGTEXT
```

The text data types also have the same size limitations and storage requirements as the binary data types. If you refer to the previous table, you can see how the sizes correspond between the binary data types and the text data types. The main difference between the two types is that the text data types are associated with a specific character set. Binary columns are treated as strings, and sorting is

case sensitive. Text columns are treated according to their character sets, and sorting is based on the collation for that character set.

The following `CREATE TABLE` statement provides an example of a `TEXT` column named DescriptionDoc:

```
CREATE TABLE Catalog
(
   ProductID SMALLINT UNSIGNED,
   Name VARCHAR(40),
   DescriptionDoc TEXT CHARACTER SET latin1 COLLATE latin1_bin
);
```

As you can see, the DescriptionDoc column includes a `CHARACTER SET` and `COLLATE` clause. The `latin1` character set and the `latin1_bin` collation are specific to the values in the DescriptionDoc column. The advantage of this is that you can use a character set and collation that differ from that of the table, database, or server.

Now take a look at the list data types, which are the last set of string data types. As the following syntax shows, the list data types include `ENUM` and `SET`:

```
<list data type>::=
{ENUM | SET} (<value> [{, <value>}...])
```

The `ENUM` data type allows you to specify a list of values that you can use in a column configured with that type. When you insert a row in the table, you can also insert one of the values defined for the data type in the column. The column can contain only one value, and it must be one of the listed values. A `SET` data type also specifies a list of values to be inserted in the column. Unlike the `ENUM` data type, in which you can specify only one value, the `SET` data type allows you to specify multiple values from the list.

The following table definition illustrates how you can configure an `ENUM` column and a `SET` column:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED,
   BikeModel VARCHAR(40),
   BikeColor ENUM('red', 'blue', 'green', 'yellow'),
   BikeOptions SET('rack', 'light', 'helmet', 'lock')
);
```

Notice that the list of values follows the data type. The values are enclosed in single quotes and separated by commas, and all values are enclosed in parentheses. For an `ENUM` data type, you can specify up to 65,535 values. For a `SET` data type, you can specify up to 64 values.

Date/Time Data Types

The final category of data types is the date/time data types, which are shown in the following syntax:

```
<date/time data type>::=
DATE | TIME | DATETIME | YEAR | TIMESTAMP
```

The date/time data types allow you to specify columns that contain data specific to dates and times. The date/time data types support the ranges and formats shown in the following table:

Data type	Format	Range
DATE	YYYY-MM-DD	1000-01-01 through 9999
TIME	HH:MM:SS	–838:59:59 to 838:59:59
DATETIME	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 through 9999
YEAR	YYYY	1901 to 2155 (and 0000)
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1970-01-01 00:00:00 to partway through 2037

The date/time data types are handy if you want to store specific types of date information. For example, if you want to record only the date and not the year, you would use the `DATE` data type. The values that you entered in this column would have to conform to the format defined by that data type. However, of particular interest is the `TIMESTAMP` data type, which is slightly different from the other data types. When you configure a column with this data type, a row, when inserted or updated, is automatically provided a value for the `TIMESTAMP` column that is based on the current time and date. This provides a handy way to record each transaction that occurs in a particular table.

Now take a look at a table that uses time/date data types. The following table definition includes a `YEAR` column and a `TIMESTAMP`:

```
CREATE TABLE BookOrders
(
   OrderID SMALLINT UNSIGNED,
   BookID SMALLINT UNSIGNED,
   Copyright YEAR,
   OrderDate TIMESTAMP
);
```

The Copyright column allows you to add a value to the column that falls in the range of 1901 to 2155; however, you're restricted from adding any other types of values. The OrderDate column automatically records the current data and time when a particular row is inserted or updated, so you don't have to insert any values in this column.

Defining a Column's Nullability

Up to this point, the focus has been on identifying only the required elements of a `CREATE TABLE` statement and the column definitions. As a result, the column definitions have included only the column names and the data types assigned to those columns. The next component of the column definition examined in this chapter is the column's nullability, which is specified through the `NULL` and `NOT NULL` keywords.

A column's nullability refers to a column's ability to accept null values. Recall from Chapter 1 that a null value indicates that a value is undefined or unknown. It is not the same as zero or blank, but instead means that the value is absent. When you include `NOT NULL` in your column definition, you're saying that the column does not permit null values. In other words, a specific value must always be provided for that column. On the other hand, the `NULL` option permits null values. If neither option is specified, `NULL` is assumed, and null values are permitted in the column.

Now take a look at a table definition that specifies the nullability of its columns. The following example creates the Catalog table and includes two NOT NULL columns:

```
CREATE TABLE Catalog
(
   ProductID SMALLINT UNSIGNED NOT NULL,
   Name VARCHAR(40) NOT NULL
);
```

You must provide a value for both the ProductID column and the Name column. Whenever you insert rows in the table or update rows in this table, you cannot use NULL as a value for either of those columns. In general, whenever you configure a column as NOT NULL, you must supply a value other than NULL to the column when inserting and modifying rows. There are, however, two exceptions to this rule. If you configure a column with the TIMESTAMP data type or if you use the AUTO_INCREMENT option, then inserting NULL automatically sets the value of the column to the correct TIMESTAMP value or the AUTO_INCREMENT value. (The AUTO_INCREMENT option is discussed later in the chapter.) But other than these two exceptions, a NOT NULL column cannot accept a null value.

Defining Default Values

Situations might arise in which you want a column to use a default value when inserting or updating a row, if no value is provided for a column. This is useful when a value is often repeated in a column or it is the value most likely to be used in that column. MySQL allows you to assign default values through the use of a DEFAULT clause. For example, the following table definition includes a column defined with a default value of Unknown:

```
CREATE TABLE AuthorBios
(
   AuthID SMALLINT UNSIGNED NOT NULL,
   YearBorn YEAR NOT NULL,
   CityBorn VARCHAR(40) NOT NULL DEFAULT 'Unknown'
);
```

In this CREATE TABLE statement, the CityBorn column is configured with the VARCHAR data type and the NOT NULL option. In addition, the column definition includes a DEFAULT clause. In that clause, the keyword DEFAULT is specified, followed by the actual default value, which in this case is Unknown. If you insert a row in the table and do not specify a value for the CityBorn column, the value Unknown is automatically inserted in that column.

You can also specify a default value in a column configured with a numeric data type. In the following table definition, the NumBooks column is configured with the SMALLINT data type and a default value of 1:

```
CREATE TABLE AuthorBios
(
   AuthID SMALLINT UNSIGNED NOT NULL,
   YearBorn YEAR NOT NULL,
   NumBooks SMALLINT NOT NULL DEFAULT 1
);
```

Notice that you do not need to enclose the default value in single quotes. The quote marks are used only for defaults that are string values.

You can also specify `NULL` *as a default value. The column, though, must permit null values in order to specify it as a default.*

If you do not assign a default value to a column, MySQL automatically assigns a default to the column. If a column accepts null values, the default is `NULL`. If a column does not accept null values, the default depends on how the column is defined:

- For columns configured with the `TIMESTAMP` data type, the default value for the first `TIMESTAMP` column is the current date and time. The default values for any other `TIMESTAMP` columns in the table are zero values in place of the date and time.
- For columns configured with a date/time data type other than `TIMESTAMP`, the default values are zero values in place of the date and time.
- For columns configured with the `AUTO_INCREMENT` option, the default value is the next number in the incremented sequence of numbers. (The `AUTO_INCREMENT` option is discussed later in the chapter.)
- For numeric columns that are not configured with the `AUTO_INCREMENT` option, the default value is 0.
- For columns configured with the `ENUM` data type, the default value is the first value specified in the column definition.
- For columns configured with a string data type other than the `ENUM` type, the default value is an empty string.

As you can see, the rules for defaults in `NOT NULL` columns are more complex than for columns that permit null values. As a result, you might consider defining defaults on any columns whose default value you want to control.

Most relational database management systems (RDBMSs) do not automatically assign default values to all columns. For these systems, trying to insert a value in a column for which you defined no default and null values are not permitted results in an error. As you can see with MySQL, all columns are assigned a default value.

Defining Primary Keys

In Chapter 4, when you were learning how to create a data model, you were introduced to the concept of primary keys and how they ensure the uniqueness of each row in a table. A primary key is one or more columns in a table that uniquely identify each row in that table. For nearly any table you create, you should define a primary key for that table.

The easiest way to define a single-column primary key is to specify the `PRIMARY KEY` option in the column definition, as shown in the following example:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL,
   ModelDescrip VARCHAR(40)
);
```

In this table definition, the primary key for the Orders table is defined on the OrderID column. You only need to add the PRIMARY KEY clause to the column definition. In order to define a column as a primary key, the column must be configured as NOT NULL. If you do not explicitly specify the NOT NULL option, NOT NULL is assumed. In addition, a table can have only one primary key, which you can define in the column definition or as a separate constraint, as shown in the following syntax:

```
[CONSTRAINT <constraint name>] PRIMARY KEY (<column name> [{, <column name>}...])
```

When you define a primary key as a separate constraint, you're including it as a table element in your table definition, as you would other table elements, such as column definitions. For example, you can define the same primary key that is shown in the preceding example as a table element. The following table definition for the Orders table removes the PRIMARY KEY clause from the OrderID column definition and uses a PRIMARY KEY constraint:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL,
   ModelID SMALLINT UNSIGNED NOT NULL,
   ModelDescrip VARCHAR(40),
   PRIMARY KEY (OrderID)
);
```

As you can see, the PRIMARY KEY constraint is added as another table element, just like the three columns. The table element needs to include only the keywords PRIMARY KEY and the name of the primary key column, enclosed in parentheses. If you were creating a primary key on more than one column, you would include both of those column names in the parentheses, as shown in the following table definition:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL,
   ModelID SMALLINT UNSIGNED NOT NULL,
   ModelDescrip VARCHAR(40),
   PRIMARY KEY (OrderID, ModelID)
);
```

Notice that the PRIMARY KEY constraint now specifies the OrderID column and the ModelID column. As a result, the primary key for the Orders table will be created on these two columns, which means that no two value pairs can be alike, although values can be repeated in individual columns. Any time that you plan to define a primary key on two or more columns, you must use the table-level constraint. You cannot define a primary key on multiple columns at the column level, as you can when you include only one column in the primary key.

Defining Auto-Increment Columns

In some cases, you may want to generate the numbers in your primary key automatically. For example, each time you add an order to a new table, you want to assign a new number to identify that order. The more rows that the table contains, the more order numbers there will be. For this reason, MySQL allows you to define a primary key column with the AUTO_INCREMENT option. The AUTO_INCREMENT option allows you to specify that numbers be generated automatically for your foreign key column. For example, the primary key column in the following example is configured with the AUTO_INCREMENT option:

```
CREATE TABLE Catalog
(
   ProductID SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
   Name VARCHAR(40) NOT NULL,
   PRIMARY KEY (ProductID)
);
```

In this example, the ProductID column is configured with the `SMALLINT` data type and is configured as the primary key (through the use of a `PRIMARY KEY` constraint). The column definition also includes the `NOT NULL` option and the `AUTO_INCREMENT` option. As a result, whenever you add a new row to the Catalog table, a new number is automatically assigned to the ProductID column. The number is incremented by 1, based on the highest value existing in that column. For example, if a row exists with a ProductID value of 1347, and this is the highest ProductID value in the table, the next row inserted in the table is assigned a ProductID value of 1348.

You can use the `AUTO_INCEREMENT` option only on a column configured with an integer data type and the `NOT NULL` option. In addition, the table must be set up as a primary key or with a unique index, and there can be only one `AUTO_INCREMENT` column per table. (Unique indexes are discussed later in the chapter.) Also, the `AUTO_INCREMENET` column cannot be defined with a default value.

Defining Foreign Keys

In Chapter 4, you learned how tables in a relational database form relationships with each other in order to associate data in a meaningful way and to help ensure data integrity. When you created your data model, you showed these relationships by connecting related tables with lines that indicated the type of relationship. In your final data model, you found that the most common type of relationship was the one-to-many, which was represented by a line that had three prongs on the *many* side of the relationship.

In order to implement these relationships in MySQL, you must define foreign keys on the referencing tables. You define the foreign key on the column or columns in the table that references the column or columns in the referenced table. The referencing table, the table that contains the foreign key, is often referred to as the *child* table, and the referenced table is often referred to as the *parent* table.

The foreign key maintains the consistency of the data between the child table the parent table. In order to insert a row in the child table or to update a row in that table, the value in the foreign key column must exist in the referenced column in the parent table.

For example, suppose you have a table that tracks sales for a bookstore. One of the columns in the table stores the IDs for the books that have sold. Data about the books themselves is actually stored in a separate table, and each book is identified in that table by its ID. As a result, the book ID column in the sales table references the book ID column in the books table. To associate the data in these two columns, the book ID column in the sales table is configured as a foreign key. Because of the foreign key, no book ID can be added to the sales table that doesn't exist in the books table. The sales table, then, is the child table, and the books table is the parent table.

To add a foreign key to a table, you can define it in the column definition or as a constraint in a separate table element. To add the foreign key to a column definition, you must add a reference definition, which is shown in the following syntax:

```
<reference definition>::=
REFERENCES <table name> [(<column name> [{, <column name>}...])]
[ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
[ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
```

As you can see from the syntax, the clause includes several required elements: the `REFERENCES` keyword, the name of the referenced (parent) table, and at least one column in that table, enclosed in parentheses. The syntax also includes an optional `ON DELETE` clause and an optional `ON UPDATE` clause. The `ON DELETE` clause specifies how MySQL treats related data in the child table when a row in the parent table is deleted. The `ON UPDATE` clause specifies how MySQL treats related data in the child table when a row in the parent table is updated. For each clause, five options are available. You can specify only one option for each clause. These options are described in the following table:

Option	Description
RESTRICT	If the child table contains values in the referencing columns that match values in the referenced columns in the parent table, rows in the parent table cannot be deleted, and values in the referenced columns cannot be updated. This is the default option if an ON DELETE or ON UPDATE clause is not specified.
CASCADE	Rows in the child table that contain values that also exist in the referenced columns of the parent table are deleted when the associated rows are deleted from the parent table. Rows in the child table that contain values that also exist in the referenced columns of the parent table are updated when the associated values are updated in the parent table.
SET NULL	Values in the referencing columns of the child table are set to NULL when rows with referenced data in the parent table are deleted from the parent table or when the referenced data in the parent table is updated. To use this option, all referencing columns in the child table must permit null values.
NO ACTION	No action is taken in the child table when rows are deleted from the parent table or values in the referenced columns in the parent table are updated.
SET DEFAULT	Values in the referencing columns of the child table are set to their default values when rows are deleted from the parent table or the referenced columns of the parent table are updated.

When you define a foreign key, you can include an `ON DELETE` clause, an `ON UPDATE` clause, or both. If you include both, you can configure them with the same option or with different options. If you exclude one or both, the `RESTRICT` option is assumed in either case, which means that updates and deletes are limited to rows with nonreferenced values. In addition, when defining a foreign key, the referencing columns must have data types compatible with the referenced columns. For integer data types, the size and signed/unsigned status must be the same. The length of a string data type, however, doesn't have to be the same. It's generally a good idea to configure the referencing and referenced columns with the same data type and type-related options.

Now that you have an overview of how to use a reference definition to add a foreign key to a column definition, take a look at an example to help demonstrate how this works. In the following `CREATE TABLE` statement, a reference definition has been added to the ModelID column:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL REFERENCES Models (ModelID),
   ModelDescrip VARCHAR(40)
);
```

In this example, the ModelID column is configured with a `SMALLINT` data type (unsigned), a `NOT NULL` option, and a `REFERENCES` clause, which specifies the name of the parent table (Models) and the name of the referenced column (ModelID) in the parent table. As a result, the ModelID column of the Orders table can include only values that are listed in the ModelID column of the Models table.

You can also define this foreign key as a separate table element by adding a `FOREIGN KEY` constraint to your table definition. The following syntax shows how to define a `FOREIGN KEY` constraint:

```
[CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
     (<column name> [{, <column name>}...]) <reference definition>
```

As the syntax indicates, you must include the keywords `FOREIGN KEY`, the name of the referencing columns in the child table, enclosed in parentheses, and a reference definition. The reference definition is the same definition used in a column definition to add a foreign key. To illustrate this, rewrite the last example table definition, but this time use a `FOREIGN KEY` constraint to define the foreign key:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL,
   ModelDescrip VARCHAR(40),
   FOREIGN KEY (ModelID) REFERENCES Models (ModelID)
      ON DELETE CASCADE ON UPDATE CASCADE
);
```

In this example, the `FOREIGN KEY` constraint is added as a table element, along with the column definitions. The same column (ModelID) is being configured as a foreign key that references the ModelID column of the Models table. The only difference between this example and the last example is that the reference definition in the last example includes an `ON DELETE` clause and an `ON UPDATE` clause, both of which are configured with the `CASCADE` option. As a result, the child table reflects changes to the parent table.

If you want to define a foreign key on more than one column, you must use a `FOREIGN KEY` constraint, rather than adding a referencing definition to the column definition. In addition, you must separate the column names by commas and enclose all the column names in parentheses.

Defining Table Types

When you were first introduced to the table definition syntax earlier in the chapter, one of the last elements in that syntax was the `<table option>` placeholder. For each table definition, you can include one or more table options.

> *For the most part, these options are beyond the scope of this book. If you want to learn more about them, you are encouraged to review the MySQL product documentation.*

One of the table options that is especially important when learning about MySQL is the one that allows you to define the type of table that you create in your table definition. Recall from Chapter 3 that MySQL allows you to create six different types of tables, which are shown in the following syntax:

```
ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MYISAM}
```

To define a table type, you must include an `ENGINE` clause at the end of your table definition, after the parentheses that enclose your table elements. For example, the following table definition specifies the `InnoDB` table type:

```
CREATE TABLE AuthorBios
(
   AuthID SMALLINT UNSIGNED NOT NULL,
   YearBorn YEAR NOT NULL,
   CityBorn VARCHAR(40) NOT NULL DEFAULT 'Unknown'
)
ENGINE=INNODB;
```

In this definition, an `ENGINE` clause is added after the last column definition and closing parentheses. Notice that you simply specify the `ENGINE` keyword, the equal sign, and one of the seven table types. Each table type in MySQL supports a specific set of functionality and serves specific purposes. In addition, each type is associated with a related storage engine (handler) that processes the data in that table. For example, the `MyISAM` engine processes data in `MyISAM` tables. The following table discusses each of the six types of tables.

Table type	Description
BDB	A transaction-safe table that is managed by the Berkeley DB (BDB) handler. The BDB handler also supports automatic recovery and page-level locking. The BDB handler does not work on all the operating systems on which MySQL can operate. For the most part, InnoDB tables have replaced BDB tables.
MEMORY	A table whose contents are stored in memory. The data stored in the tables is available only as long as the MySQL server is available. If the server crashes or is shut down, the data disappears. Because these types of tables are stored in memory, they are very fast and are good candidates for temporary tables. MEMORY tables can also be referred to as HEAP tables, although MEMORY is now the preferable keyword.
InnoDB	A transaction-safe table that is managed by the InnoDB handler. As a result, data is not stored in a .MYD file, but instead is managed in the InnoDB tablespace. InnoDB tables also support full foreign key functionality in MySQL, unlike other tables. In addition, the InnoDB handler supports automatic recovery and row-level locking. InnoDB tables do not perform as well as MyISAM tables.
ISAM	A deprecated table type that was once the default table type in MySQL. The MyISAM table type has replaced it, although it is still supported for backward compatibility. Eventually, ISAM tables will no longer be supported.

Table continued on following page

MERGE	A virtual table that is made up of identical MyISAM tables. Data is not stored in the MERGE table, but in the underlying MyISAM tables. Changes made to the MERGE table definition do not affect the underlying MyISAM tables. MERGE tables can also be referred to as MRG_MyISAM tables
MyISAM	The default table type in MySQL. MyISAM tables, which are based on and have replaced ISAM tables, support extensive indexing and are optimized for compression and speed. Unlike other table types, BLOB and TEXT columns can be indexed and null values are allowed in indexed columns. MyISAM tables are not transaction safe, and they do not support full foreign key functionality.

You can use the `TYPE` *keyword to specify the table type, rather than the* `ENGINE` *keyword. However,* `TYPE` *has been deprecated in MySQL, which means that it will eventually be phased out. If you use* `TYPE`*, you receive a warning about its deprecated state, but the table is still created.*

Creating Tables in the DVDRentals Database

Now that you have learned how to create a table in MySQL, it's time to try it out for yourself. The following three Try It Out sections walk you through the steps necessary to create the tables in the DVDRentals database. The tables are based on the final database design that you developed in Chapter 4. The tables are divided into three categories that correspond to the following Try It Out sections. The first group of tables acts as lookup tables in the database. They must be developed before you create tables that reference the lookup table. The second category of tables holds data about the people who will participate somehow in the database system. These include the movie participants, the employees, and the customers. The last group includes the tables that contain foreign keys. You must create this group of tables last because they contain columns that reference other tables. In addition, you must create these tables in a specific order because of how they reference each other.

In this Try It Out, you create the six lookup tables that are part of the DVDRentals database, which you created in the Try It Out section earlier in the chapter. These tables include the Roles, MovieTypes, Studios, Ratings, Formats, and Status tables. As you work your way through this exercise, you should reference the data model that you created in Chapter 4. From there, you can compare the SQL statement that you use here to that model.

Try It Out Creating the Lookup Tables

Follow these steps to create the six lookup tables:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To create the Roles table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Roles
   (
   RoleID VARCHAR(4) NOT NULL,
   RoleDescrip VARCHAR(30) NOT NULL,
   PRIMARY KEY (RoleID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

3. To create the MovieTypes table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE MovieTypes
   (
   MTypeID VARCHAR(4) NOT NULL,
   MTypeDescrip VARCHAR(30) NOT NULL,
   PRIMARY KEY (MTypeID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

4. To create the Studios table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Studios
   (
   StudID VARCHAR(4) NOT NULL,
   StudDescrip VARCHAR(40) NOT NULL,
   PRIMARY KEY (StudID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

5. To create the Ratings table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Ratings
   (
   RatingID VARCHAR(4) NOT NULL,
   RatingDescrip VARCHAR(30) NOT NULL,
   PRIMARY KEY (RatingID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

6. To create the Formats table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Formats
   (
   FormID CHAR(2) NOT NULL,
   FormDescrip VARCHAR(15) NOT NULL,
   PRIMARY KEY (FormID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

7. To create the Status table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Status
   (
   StatID CHAR(3) NOT NULL,
   StatDescrip VARCHAR(20) NOT NULL,
   PRIMARY KEY (StatID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

How It Works

In this exercise, you created the six lookup tables in the DVDRentals database. The table definitions should be consistent with the final data model that you created in Chapter 4 for the DVDRentals database. In addition, the six tables are very similar. Take a look at one of them, and review the code that you used to create the table. You used the following `CREATE TABLE` statement to create the Roles table:

```
CREATE TABLE Roles
   (
   RoleID VARCHAR(4) NOT NULL,
   RoleDescrip VARCHAR(30) NOT NULL,
   PRIMARY KEY (RoleID)
   )
ENGINE=INNODB;
```

The statement begins with the `CREATE TABLE` statement, which identifies the name of the new table (Roles). The table definition then includes three table elements, which are separated by commas and enclosed in parentheses. The first two table elements are column definitions. The RoleID column definition creates a column that is configured with a `VARCHAR` data type. The data type permits up to four characters. In addition, the column does not permit null values. You use a `VARCHAR` data type for the RoleID column, rather than a `CHAR` data type because MySQL converts `CHAR` data types to `VARCHAR` data types whenever more than three characters are specified for the value length *and* there are other varying-length columns in the table (which is the case for the RoleDescrip column). Otherwise, you would use `CHAR(4)` because the values in the column have a fixed length of four characters.

The second column defined in the Roles table definition is the RoleDescrip column, which is configured with a `VARCHAR` data type and a maximum length of 30 characters. This column also does not permit null values. The last table element in the `CREATE TABLE` statement is the `PRIMARY KEY` constraint, which defines a primary key on the RoleID column. As a result, this column uniquely identifies each role in the table.

The last component of the Roles table definition is the `ENGINE` table option, which species that the table type is `InnoDB`. You specify this table type because `InnoDB` is the only type that supports transactions and foreign keys, both of which are important to the DVDRentals database.

The other five tables that you created in the exercise are nearly identical to the Roles table, except for the names of the tables and columns. The only other difference is that string columns with a length less than four are configured with `CHAR` data types rather than `VARCHAR`.

Once you create the six lookup tables, you can create the three tables that contain the people (explained in the following Try It Out). Because these tables do not contain foreign keys that reference other tables, you could have created these three tables first. In fact, you could have created the nine tables in any order, as long as all referenced (parent) tables are created before the referencing (child) tables. You grouped the tables together in the manner you did just to keep similar types of tables together in order to make explanations simpler.

Try It Out Creating the People Tables

The following steps describe how to create the three tables that contain people:

1. To create the Participants table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Participants
   (
   PartID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   PartFN VARCHAR(20) NOT NULL,
   PartMN VARCHAR(20) NULL,
   PartLN VARCHAR(20) NULL
   )
ENGINE=INNODB;
```

   You should receive a message indicating that the statement executed successfully.

2. To create the Employees table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Employees
   (
   EmpID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   EmpFN VARCHAR(20) NOT NULL,
   EmpMN VARCHAR(20) NULL,
   EmpLN VARCHAR(20) NOT NULL
   )
ENGINE=INNODB;
```

   You should receive a message indicating that the statement executed successfully.

3. To create the Customers table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Customers
   (
   CustID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   CustFN VARCHAR(20) NOT NULL,
   CustMN VARCHAR(20) NULL,
   CustLN VARCHAR(20) NOT NULL
   )
ENGINE=INNODB;
```

   You should receive a message indicating that the statement executed successfully.

How It Works

As with the previous exercise, this exercise adds several tables to the DVDRentals database. Except for the names of the tables and columns, the table definitions are nearly identical. As a result, this explanation covers only one of these definitions to understand how the statements work. The following `CREATE TABLE` statement is the one you used to create the Participants table:

```
CREATE TABLE Participants
   (
   PartID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   PartFN VARCHAR(20) NOT NULL,
   PartMN VARCHAR(20) NULL,
   PartLN VARCHAR(20) NULL
   )
ENGINE=INNODB;
```

The Participants table definition includes four table elements, separated by commas and enclosed in parentheses. All four table elements are column definitions. The first column definition defines the PartID column, which is configured with the `SMALLINT` data type, the `NOT NULL` option, and the `AUTO_INCREMENT` option. The column is also defined as the primary key. As a result, values in the column uniquely identify each row in the table, null values are not allowed, and the values inserted in the column are generated automatically.

The remaining three columns are configured with the `VARCHAR` data type and are assigned a length of 20 characters. Null values are not allowed in the PartFN column, but they are allowed in the PartMN columns and the PartLN columns. The columns are set up this way to allow for actors and other movie participants who are known by only one name. (Cher comes to mind as one example.) The other two tables — Employees and Customers — are different in this respect because a last name is required. This, of course, is a business decision, and the business rules collected for this project would dictate which names are actually required. In the case of these three tables, a middle name is not required for any of them.

As with the six lookup table that you created in the previous exercise, all three of the tables in this exercise have been created as `InnoDB` tables. To support foreign key functionality, all tables participating in relationships must be configured as `InnoDB` tables.

Now that you've created all the referenced tables in the DVDRentals database, you're ready to create the referencing tables, which are each configured with one or more foreign keys. The order in which you create these remaining four tables is important because dependencies exist among these four tables. For example, you must create the DVDs table before you create the DVDParticipant and Transactions tables because both these tables reference the DVDs table. In addition, you must create the Orders table before you create the Transactions table because the Transactions table references the Orders table. The following Try It Out shows you how to create all the necessary foreign key tables.

Try It Out — Creating the Foreign Key Tables

The following steps describe how to create the four referencing tables:

1. To create the DVDs table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE DVDs
   (
   DVDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   DVDName VARCHAR(60) NOT NULL,
   NumDisks TINYINT NOT NULL DEFAULT 1,
   YearRlsd YEAR NOT NULL,
   MTypeID VARCHAR(4) NOT NULL,
   StudID VARCHAR(4) NOT NULL,
   RatingID VARCHAR(4) NOT NULL,
   FormID CHAR(2) NOT NULL,
   StatID CHAR(3) NOT NULL,
   FOREIGN KEY (MTypeID) REFERENCES MovieTypes (MTypeID),
   FOREIGN KEY (StudID) REFERENCES Studios (StudID),
   FOREIGN KEY (RatingID) REFERENCES Ratings (RatingID),
   FOREIGN KEY (FormID) REFERENCES Formats (FormID),
   FOREIGN KEY (StatID) REFERENCES Status (StatID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

This table definition is based on MySQL version 4.1 or later. This statement will not work for versions earlier than 4.1.

2. To create the DVDParticipant table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE DVDParticipant
   (
   DVDID SMALLINT NOT NULL,
   PartID SMALLINT NOT NULL,
   RoleID VARCHAR(4) NOT NULL,
   PRIMARY KEY (DVDID, PartID, RoleID),
   FOREIGN KEY (DVDID) REFERENCES DVDs (DVDID),
   FOREIGN KEY (PartID) REFERENCES Participants (PartID),
   FOREIGN KEY (RoleID) REFERENCES Roles (RoleID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

3. To create the Orders table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Orders
   (
   OrderID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   CustID SMALLINT NOT NULL,
   EmpID SMALLINT NOT NULL,
   FOREIGN KEY (CustID) REFERENCES Customers (CustID),
   FOREIGN KEY (EmpID) REFERENCES Employees (EmpID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

4. To create the Transactions table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Transactions
   (
   TransID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   OrderID INT NOT NULL,
   DVDID SMALLINT NOT NULL,
   DateOut DATE NOT NULL,
   DateDue DATE NOT NULL,
   DateIn DATE NOT NULL,
   FOREIGN KEY (OrderID) REFERENCES Orders (OrderID),
   FOREIGN KEY (DVDID) REFERENCES DVDs (DVDID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

How It Works

Because you created the DVDs table first, that is the first table reviewed. The following `CREATE TABLE` statement creates a table definition that includes 14 table elements:

```
CREATE TABLE DVDs
   (
   DVDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   DVDName VARCHAR(60) NOT NULL,
   NumDisks TINYINT NOT NULL DEFAULT 1,
   YearRlsd YEAR NOT NULL,
   MTypeID VARCHAR(4) NOT NULL,
   StudID VARCHAR(4) NOT NULL,
   RatingID VARCHAR(4) NOT NULL,
   FormID CHAR(2) NOT NULL,
   StatID CHAR(3) NOT NULL,
   FOREIGN KEY (MTypeID) REFERENCES MovieTypes (MTypeID),
   FOREIGN KEY (StudID) REFERENCES Studios (StudID),
   FOREIGN KEY (RatingID) REFERENCES Ratings (RatingID),
   FOREIGN KEY (FormID) REFERENCES Formats (FormID),
   FOREIGN KEY (StatID) REFERENCES Status (StatID)
   )
ENGINE=INNODB;
```

As you can see, the DVDs table definition includes nine columns. Each column is configured with a data type appropriate to that column. Any column defined as a foreign key is configured with a data type identical to the referenced column. In addition, every column is configured with the `NOT NULL` option, which means that null values are not permitted. The first column, DVDID, is defined as the primary key and includes the `AUTO_INCREMENT` option, so unique values are automatically assigned to that column.

Of particular interest in this table definition is the NumDisks column definition, which includes a `DEFAULT` clause (with a default value of 1). As a result, whenever a row is inserted in the table, the value

for the NumDisks column is set to 1, unless otherwise specified. This was done because most DVDs come with one disk, although some include more.

The table definition also includes five `FOREIGN KEY` constraints, one for each referencing column. In each case, the constraint specifies the referencing column, the referenced table, and the referenced column. For example, the first `FOREIGN KEY` constraint specifies the MTypeID column as the referencing column, the MovieTypes table as the referenced table, and the MTypeID column in the MovieTypes table as the referenced column.

The other three tables that you defined in this exercise include column and foreign key definitions similar to what you've seen in the DVD table definition and the table definitions in the previous two exercises. In addition, all four tables are defined as `InnoDB` tables to support transactions and foreign key functionality. The DVDParticipant table definition includes an element that you have not seen, so that definition is worth a closer look:

```
CREATE TABLE DVDParticipant
   (
   DVDID SMALLINT NOT NULL,
   PartID SMALLINT NOT NULL,
   RoleID VARCHAR(4) NOT NULL,
   PRIMARY KEY (DVDID, PartID, RoleID),
   FOREIGN KEY (DVDID) REFERENCES DVDs (DVDID),
   FOREIGN KEY (PartID) REFERENCES Participants (PartID),
   FOREIGN KEY (RoleID) REFERENCES Roles (RoleID)
   )
ENGINE=INNODB;
```

In this table definition, a composite primary key is defined on the DVDID, PartID, and RoleID columns, all of which are configured as individual foreign keys. As this table demonstrates, primary keys can consist of multiple columns, and those columns can also be configured as foreign keys. Because the three columns, when taken as a whole, uniquely identify each row in the table, you do not have to create an additional column in order to create a primary key. The table, as it exists here, is complete.

Once you create the four remaining tables in the DVDRentals database, you can begin adding the data necessary to populate the tables. As you learned earlier in this chapter, the data must exist in the referenced columns before you can insert it in the referencing columns. What this means is that the lookup tables and the tables that contain people's names must be populated before the other tables. Chapter 6 provides more detail on how you insert data in your tables, but for now, the focus switches to modifying table definitions.

Modifying Tables

It is not uncommon to find that, after creating a table, you want to modify the table definition. Fortunately, MySQL allows you to change a number of table elements after creating a table. For example, you can add columns, alter existing columns, add `PRIMARY KEY` and `FOREIGN KEY` constraints, or remove columns and constraints.

To modify an existing table definition, you must use the `ALTER TABLE` statement. The following syntax shows how to create an `ALTER TABLE` statement and the options available to that statement:

```
ALTER TABLE <table name>
<alter option> [{, <alter option>}...]

<alter option>::=
{ADD [COLUMN] <column definition> [FIRST | AFTER <column name>]}
| {ADD [COLUMN] (<table element> [{, <table element>}...])}
| {ADD [CONSTRAINT <constraint name>] PRIMARY KEY
     (<column name> [{, <column name>}...])}
| {ADD [CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
     (<column name> [{, <column name>}...]) <reference definition>}
| {ADD [CONSTRAINT <constraint name>] UNIQUE [<index name>]
     (<column name> [{, <column name>}...])}
| {ADD INDEX [<index name>] (<column name> [{, <column name>}...])}
| {ADD FULLTEXT [<index name>] (<column name> [{, <column name>}...])}
| {ALTER [COLUMN] <column name> {SET DEFAULT <value> | DROP DEFAULT}}
| {CHANGE [COLUMN] <column name> <column definition> [FIRST | AFTER <column name>]}
| {MODIFY [COLUMN] <column definition> [FIRST | AFTER <column name>]}
| {DROP [COLUMN] <column name>}
| {DROP PRIMARY KEY}
| {DROP INDEX <index name>}
| {DROP FOREIGN KEY <constraint name>}
| {RENAME [TO] <new table name>}
| {ORDER BY <column name> [{, <column name>}...]}
| {<table option> [<table option>...]}
```

The basic elements of the `ALTER TABLE` statement are the `ALTER TABLE` keywords, the name of the table that you want to modify, and one or more alter options. If you chose more than one option, you must separate the options with a comma. Each of the alter options maps directly to a table definition option, except that you must also include an action keyword such as `ADD`, `ALTER`, or `DROP`. In addition, several of the alter options include additional elements that help to define the option. Take a look at an example to help illustrate this concept. Suppose that you create the following table:

```
CREATE TABLE Books
(
   BookID SMALLINT NOT NULL,
   BookName VARCHAR(40) NOT NULL,
   PubID SMALLINT NOT NULL DEFAULT 'Unknown'
)
ENGINE=INNODB;
```

As you can see, the table definition creates a table named Books, the table contains three columns, and the PubID column contains a default value of Unknown. Now suppose that you want to modify the table to include a primary key, foreign key, and an additional column. The following `ALTER TABLE` statement modifies the table accordingly:

```
ALTER TABLE Books
ADD PRIMARY KEY (BookID),
ADD CONSTRAINT fk_1 FOREIGN KEY (PubID) REFERENCES Publishers (PubID),
ADD COLUMN Format ENUM('paperback', 'hardcover') NOT NULL AFTER BookName;
```

The statement begins with the ALTER TABLE statement, which identifies the name of the table being modified, which in this case is Books. The next line adds a primary key to the table. The primary key is based on the BookID column. The third line in the ALTER TABLE statement adds a FOREIGN KEY constraint to the table. The name of the constraint is fk_1, the foreign key is defined on the PubID column, and the foreign key references the PubID column in the Publishers table.

The final line of the ALTER TABLE statement adds a column to the table. As you can see, a column definition follows the ADD COLUMN keywords. The name of the column is Format. The column is configured with an ENUM data type that is defined with two values: paperback and hardcover. The column is also configured with the NOT NULL option. The AFTER clause, which is unique to the ALTER TABLE statement, specifies that the new column should be added after the column named BookName.

As you can see, the options available to the ALTER TABLE statement are very consistent to their CREATE TABLE statement counterparts, at least in terms of adding and modifying columns. If you plan to remove a component of a table, the options are much simpler, as shown in the following example:

```
ALTER TABLE Books
DROP PRIMARY KEY,
DROP FOREIGN KEY fk_1,
DROP COLUMN Format;
```

In this ALTER TABLE statement, the primary key, the fk_1 FOREIGN KEY constraint, and the Format column are all removed form the table. As these examples demonstrate, the ALTER TABLE syntax contains few elements that you haven't seen, except for the action keywords and the few options specific to the ALTER TABLE statement. In addition, these examples also demonstrate that you can modify most of the components that you define in a CREATE TABLE statement with an ALTER TABLE statement.

Deleting Tables

Deleting a table from the database is simply a matter of executing a DROP TABLE statement. As the following syntax shows, the only elements required in a DROP TABLE statement are the DROP TABLE keywords and the name of the table:

```
DROP [TEMPORARY] TABLE [IF EXISTS] <table name> [{, <table name>}...]
```

The DROP TABLE statement also includes the optional TEMPORARY keyword, which you use if you want to ensure that you drop only a temporary table and do not inadvertently drop a permanent table. The other optional element in the DROP TABLE statement is the IF EXISTS clause. If you specify the clause, you receive a warning message, rather than an error, if you try to drop a table that doesn't exist. The one other aspect of the DROP TABLE syntax to consider is the ability to add optional table names. You can use this statement to drop multiple tables, as long as you separate them by a comma.

Now that you've seen the syntax, look at an example of a DROP TABLE statement to demonstrate how one works. The following example removes a table named Books from your database:

```
DROP TABLE IF EXISTS Books;
```

As you can see, the statement includes the `DROP TABLE` keywords, along with the name of the table. In addition, the statement includes the `IF EXISTS` clause, which means that, if the table doesn't exist, you receive a warning rather than an error when trying to drop a table.

> *You cannot drop a parent table referenced in a foreign key. You must first remove the foreign key in the child table and then drop the parent table.*

In the following exercise, you create a table named InStock, alter the table definition, and then delete the table from your database.

Try It Out Altering and Dropping Tables

To perform the tasks mentioned here, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
use test
```

   You should receive a message indicating that you switched to the test database.

2. To create the InStock table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE InStock
(
   ProductID SMALLINT
);
```

   You should receive a message indicating that the statement executed successfully.

3. Next, add a column, modify the ProductID column, and add a primary key. To make these changes, type the following `ALTER TABLE` statement at the mysql command prompt, and then press Enter:

```
ALTER TABLE InStock
ADD COLUMN Quantity SMALLINT UNSIGNED NOT NULL,
MODIFY ProductID SMALLINT UNSIGNED NOT NULL,
ADD PRIMARY KEY (ProductID);
```

   You should receive a message indicating that the statement executed successfully.

4. Next, drop the column and the primary key that you added in the previous step. To make these changes, type the following `ALTER TABLE` statement at the mysql command prompt, and then press Enter:

```
ALTER TABLE InStock
DROP COLUMN Quantity,
DROP PRIMARY KEY;
```

   You should receive a message indicating that the statement executed successfully.

5. Finally, remove the InStock table from the test database. To remove the table, type the following `DROP TABLE` statement at the mysql command prompt, and then press Enter:

```
DROP TABLE InStock;
```

   You should receive a message indicating that the statement executed successfully.

How It Works

In this exercise, you used a `CREATE TABLE` statement to create the InStock table in the test database. Once you created the table, you used the following `ALTER TABLE` statement to modify the InStock table definition:

```
ALTER TABLE InStock
ADD COLUMN Quantity SMALLINT UNSIGNED NOT NULL,
MODIFY ProductID SMALLINT UNSIGNED NOT NULL,
ADD PRIMARY KEY (ProductID);
```

The `ALTER TABLE` statement includes three alter options. The first one adds a column named Quantity to the InStock table. The column is configured with the `SMALLINT` data type (unsigned) and the `NOT NULL` option. The next alter option modifies the ProductID column by configuring the `SMALLINT` data type to be unsigned and by adding the `NOT NULL` option. The final alter option adds a primary key to the table. The primary key is based on the ProductID column.

The next step used the following statement to again modify the table:

```
ALTER TABLE InStock
DROP COLUMN Quantity,
DROP PRIMARY KEY;
```

This statement removes the Quantity column from the InStock table and then drops the primary key. After altering the table, you used a `DROP TABLE` statement to remove the InStock table from the database.

Managing Indexes

Earlier in the chapter, when you were introduced to the `CREATE TABLE` statement, you no doubt noticed that some of the table elements were related to indexes. An *index* is a device that MySQL uses to speed up searches and reduce the time it takes to execute complex queries. An index works under the same principles as an index you would find at the end of a book. The index provides an organized list of pointers to the actual data. As a result, when MySQL is executing a query, it does not have to scan each table in its entirety to locate the correct data, but it can instead scan the index, thus resulting in quicker and more efficient access.

Indexes, however, do have their trade-offs. First, they can affect the performance of operations that involve the modification of data in a table because the index must be updated whenever the table has been updated. In addition, indexes require additional disk space, which, for large tables, can translate to a substantial amount of storage. Despite these drawbacks, indexes play a critical role in data access, and few tables in a MySQL database are not indexed in some way.

Index Types

MySQL supports five types of indexes that can be created on a table. As you have worked your way through this chapter, you have already created two types of indexes: primary keys and foreign keys. Whenever you create a primary key or a foreign key, you are automatically creating an index on the columns specified in those keys. In fact, when you create a `FOREIGN KEY` constraint, you have the option

to provide a name for the index that is being created. If you don't provide a name, MySQL assigns a name based on the first referencing column. In addition, MySQL assigns the name `PRIMARY` to all primary key indexes.

> *When you are setting up a foreign key on columns in an* `InnoDB` *table, the referencing foreign key columns and the referenced columns in the parent table must both be indexed.*

In addition to primary key and foreign key indexes, MySQL also supports unique indexes, regular (non-unique) indexes, and full-text indexes. The following table provides an overview of each of the five types of indexes.

Index type	Description
Primary key	Requires that each value or set of values be unique in the columns on which the primary key is defined. In addition, null values are not allowed. Also, a table can include only one primary key.
Foreign key	Enforces the relationship between the referencing columns in the child table where the foreign key is defined and the referenced columns in the parent table.
Regular	A basic index that permits duplicate values and null values in the columns on which the index is defined.
Unique	Requires that each value or set of values be unique in the columns on which the index is defined. Unlike primary key indexes, null values are allowed.
Full-text	Supports full-text searches of the values in the columns on which the index is defined. A full-text index permits duplicate values and null values in those columns. A full-text index can be defined only on MyISAM tables and only on CHAR, VARCHAR, and TEXT columns.

When creating a table definition that includes indexes, you should place the primary key columns first, followed by the unique index columns, and then followed by any nonunique index columns. This process helps to optimize index performance. Later in the book, in Chapter 15, you learn more about how to use indexes to optimize query performance, but for now, take a look at how you actually create indexes on columns in a table.

Creating Indexes

MySQL supports several methods for adding indexes to a table. You can include the indexes in your column definition, you can use an `ALTER TABLE` statement to add an index to a table, or you can use the `CREATE INDEX` statement to add an index to a table.

Defining Indexes When Creating Tables

When using the `CREATE TABLE` statement to create a table, you can include a number of table elements in your statement. For example, you can include column definitions, a `PRIMARY KEY` constraint, or `FOREIGN KEY` constraints. In addition, you can define a primary key and foreign keys in your column definitions. Regardless of the method that you use to create a primary key or a foreign key, whenever you create such a key, you're automatically creating an index on the columns participating in a particular key.

It is worth noting here that, in MySQL, the keyword `KEY` *and the keyword* `INDEX` *are often used synonymously.*

Because you're already familiar with how to create primary key and foreign key indexes in a `CREATE TABLE` statement, take a look at creating unique, regular, and full-text indexes.

Creating Unique Indexes

To create a unique index, you should use a `UNIQUE` constraint, which is one of the table element options included in the `CREATE TABLE` statement. The following syntax shows you how to create a `UNIQUE` constraint:

```
[CONSTRAINT <constraint name>] UNIQUE [INDEX] [<index name>]
     (<column name> [{, <column name>}...])
```

When adding a unique index to a table definition, you need to include only the keyword `UNIQUE` and the name of the indexed column, enclosed in parentheses. If you're creating the index on more than one column, then you must separate the column names with a comma. In addition, you can include the `CONSTRAINT` keyword along with the name of the constraint, the `INDEX` keyword, or an index name. If you don't include a constraint name or an index name, MySQL provides names automatically. Whether or not you include these optional elements, the basic index is the same.

A unique index is also considered a constraint because it ensures that each value in a column is unique, in addition to indexing these values.

Now take a look at an example to demonstrate how to include a unique index in the table definition. The following `CREATE TABLE` statement defines a unique index on the OrderID and ModelID columns:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL,
   ModelID SMALLINT UNSIGNED NOT NULL,
   ModelDescrip VARCHAR(40),
   PRIMARY KEY (OrderID),
   UNIQUE (OrderID, ModelID)
);
```

The `CREATE TABLE` statement actually creates two indexes: one primary key and one unique. Notice that the OrderID column participates in two indexes and that the unique index is defined on two columns. As a result, the OrderID column can contain only unique values, and the OrderID and ModelID values, when taken together, can include only unique value pairs. The ModelID column, however, can include duplicate values.

Creating Regular (Nonunique) Indexes

There might be times when you want to index a column but you don't want to require that the values in the column be unique. For those situations you can use a regular index. As with unique indexes, you can include a regular index in a table definition by adding it as a table element, as shown in the following syntax:

```
{INDEX | KEY} [<index name>] (<column name> [{, <column name>}...])
```

When you define a regular index, you must specify the INDEX or KEY keyword and the name of the indexed column, enclosed in parentheses. If you want to index more than one column, you must separate the columns by a comma. For example, suppose that your database includes a table that lists the first name and the last name of a company's customers. You might want to create a composite index (an index on more than one column) on the column that contains the first name and the column that contains the last name so that the entire name can be easily searched.

In addition to specifying the indexed columns, you can provide a name for your index. If you don't provide a name, MySQL names the index automatically.

The following CREATE TABLE statement demonstrates how to include a regular index in a table definition:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL,
   ModelID SMALLINT UNSIGNED NOT NULL,
   PRIMARY KEY (OrderID),
   INDEX (ModelID)
);
```

This statement creates a regular index on the ModelID column. You do not need to specify any other elements to add the index. MySQL provides a name for the index automatically.

Creating Full-Text Indexes

Now take a look at how to add a full-text index to a MyISAM table. As you recall, you can add this type of index only to the CHAR, VARCHAR, or TEXT columns. The following syntax shows how you add a full-text index to a table definition:

```
FULLTEXT [INDEX] [<index name>] (<column name> [{, <column name>}...])
```

Adding a full-text index is almost identical to adding a regular index, except that you have to specify the keyword FULLTEXT. The following CREATE TABLE statement demonstrates how this works:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL,
   ModelID SMALLINT UNSIGNED NOT NULL,
   ModelName VARCHAR(40),
   PRIMARY KEY (OrderID),
   FULLTEXT (ModelName)
);
```

This example defines a full-text index on the ModelName column, which is configured with a VARCHAR data type. In addition, because you specify no table type option in this table definition, MySQL uses the default table type, which is MyISAM.

Now that you have an overview of how to add an index in a CREATE TABLE statement, you can try it out. In this exercise, you create a table named CDs. The table definition includes a regular index on the CDName column.

Try It Out Creating a Table with an Index

Follow these steps to complete this exercise:

1. Open the mysql client utility, type the following command, and press Enter:

```
use test
```

   You should receive a message indicating that you switched to the test database.

2. Next create the CDs table, which includes an index on the CDName column. To create the CDs table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE CDs
(
   CDID SMALLINT UNSIGNED NOT NULL,
   CDName VARCHAR(40) NOT NULL,
   INDEX (CDName)
);
```

   You should receive a message indicating that the statement executed successfully.

3. Next, remove the CDs table from the test database. To remove the table, type the following `DROP TABLE` statement at the mysql command prompt, and then press Enter:

```
DROP TABLE CDs;
```

   You should receive a message indicating that the statement executed successfully.

How It Works

In this exercise, you created the CDs table in the test database. To create this table, you used the following `CREATE TABLE` statement:

```
CREATE TABLE CDs
(
   CDID SMALLINT UNSIGNED NOT NULL,
   CDName VARCHAR(40) NOT NULL,
   INDEX (CDName)
);
```

The statement created a MyISAM table that includes two columns. The last table element in the table definition defines a regular index on the CDName column. After you created the table, you removed it from the database with a `DROP TABLE` statement.

This chapter covers only how to add an index to your table definition. In Chapter 15, you learn more about indexing and how indexes can be used to optimize performance.

Adding Indexes to Existing Tables

In addition to including indexes in a table definition, you can add an index to an existing table. You can use two methods to add an index: the `ALTER TABLE` statement and the `CREATE INDEX` statement.

Using the ALTER TABLE Statement

The `ALTER TABLE` statement allows you to add primary key, foreign key, unique, regular, and full-text indexes to a table. You already saw examples of this earlier in the chapter when you used the `ALTER TABLE` statement to add primary keys and foreign keys to a table. Now look at another example. Suppose that you used the following table definition to create a table named Orders:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL
);
```

As you can see, the table includes a primary key on the OrderID column, which means that this column has an index defined on it. Now suppose that you want to add a unique index to the table that is placed on both the OrderID column and the ModelID column. To do this, you would use the following `ALTER TABLE` statement:

```
ALTER TABLE Orders
ADD UNIQUE (OrderID, ModelID);
```

By adding the unique index, values in the two columns, when taken together, must be unique, although the ModelID column can still contain duplicate values.

In the following Try It Out you create a table named CDs, use an `ALTER TABLE` statement to add a full-text index, and then drop the table from the database.

Try It Out — Creating an Index with the ALTER TABLE Statement

To complete these tasks, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
use test
```

   You should receive a message indicating that you switched to the test database.

2. Next create the CDs table. To create the table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE CDs
(
   CDID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   CDName VARCHAR(40) NOT NULL
);
```

   You should receive a message indicating that the statement executed successfully.

3. Now you use an `ALTER TABLE` statement to add a `FULLTEXT` index to the CDs table. Type the following `ALTER TABLE` statement at the mysql command prompt, and then press Enter:

```
ALTER TABLE CDs
ADD FULLTEXT (CDName);
```

You should receive a message indicating that the statement executed successfully.

4. Finally, remove the CDs table from the test database. To remove the table, type the following `DROP TABLE` statement at the mysql command prompt, and then press Enter:

```
DROP TABLE CDs;
```

You should receive a message indicating that the statement executed successfully.

How It Works

After creating the CDs table in the test database, you used the following `ALTER TABLE` statement to add a full-text index to the table:

```
ALTER TABLE CDs
ADD FULLTEXT (CDName);
```

As you can see from the statement, you placed the full-text index on the CDName column. Because this column is configured with a `VARCHAR` data type, a full-text index could be supported. In addition, the table is set up as a MyISAM table because it is the default table type and you specified no other table type. MyISAM tables are the only tables that support full-text indexing. After adding the index to the table, you removed the table from the test database.

Using the CREATE INDEX Statement

The `CREATE INDEX` statement allows you to add unique, regular, and full-text indexes to a table, but not primary key or foreign key indexes. The following syntax shows you how to define a `CREATE INDEX` statement:

```
CREATE [UNIQUE | FULLTEXT] INDEX <index name>
ON <table name> (<column name> [{, <column name>}...])
```

To create a regular index, you need to include only the `CREATE INDEX` keyword, a name for the index, and an `ON` clause that specifies the name of the table and the columns to be indexed. If you want to create a unique index, you must also include the `UNIQUE` keyword, and if you want to create a full-text index, you must include the `FULLTEXT` keyword. In either case, you must place `UNIQUE` or `FULLTEXT` between the `CREATE INDEX` keywords, as in `CREATE UNIQUE INDEX` or `CREATE FULLTEXT INDEX`.

To demonstrate how to use a `CREATE INDEX` statement, take a look at an example based on the following table definition:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL
);
```

The `CREATE TABLE` statement shown here creates a table named Orders that contains the OrderID column and the ModelID column. The OrderID column is configured with a primary key. To add a regular index to the ModelID column, you can use the following `CREATE INDEX` statement:

```
CREATE INDEX index_1 ON Orders (ModelID);
```

Executing this statement creates a regular index named `index_1` on the ModelID column of the Orders table.

In the following exercise, you create a table named CDs, add a regular index to the table, and then drop the table from the database.

Try It Out Creating an Index with the CREATE INDEX Statement

To complete these tasks, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
use test
```

   You should receive a message indicating that you switched to the test database.

2. Next create the CDs table. To create the table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE CDs
(
   CDID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   CDName VARCHAR(40) NOT NULL
);
```

   You should receive a message indicating that the statement executed successfully.

3. Now use a `CREATE INDEX` statement to add an index to the CDs table. Type the following `CREATE INDEX` statement at the mysql command prompt, and then press Enter:

```
CREATE INDEX index_1 ON CDs (CDName);
```

   You should receive a message indicating that the statement executed successfully.

4. Finally, remove the CDs table from the test database. To remove the table, type the following `DROP TABLE` statement at the mysql command prompt, and then press Enter:

```
DROP TABLE CDs;
```

   You should receive a message indicating that the statement executed successfully.

How It Works

After you switched to the test database, you created a table named CDs. The table includes the CDID and the CDName columns, with a primary key defined on the CDID column. You then used the following `CREATE INDEX` statement to add a regular index to the table:

```
CREATE INDEX index_1 ON CDs (CDName);
```

The statement creates a regular index named `index_1` (specified in the `CREATE INDEX` statement) on the CDName column of the CDs table (specified in the `ON` clause). After adding the index, you dropped the table from the test database, which also dropped the index.

Removing Indexes

MySQL provides a couple of methods for removing an index from a table. The first of these is the ALTER TABLE statement, and the other is the DROP INDEX statement. To demonstrate how each of these statements works, take a look at the following CREATE TABLE statement:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL,
   UNIQUE unique_1 (OrderID, ModelID)
);
```

The statement defines a table named Orders. The table contains two columns: OrderID and ModelID. The table is defined with a primary key on the OrderID column and a unique index on the OrderID and ModelID columns. Now suppose that you want to drop the unique index. You could use the following ALTER TABLE statement:

```
ALTER TABLE Orders
DROP INDEX unique_1;
```

In this statement, a DROP INDEX clause drops the index named unique_1. As you can see, you merely need to specify the DROP INDEX keywords and the name of the index (in addition to the ALTER TABLE statement). You can also use a DROP INDEX statement to remove an index, as shown in the following syntax:

```
DROP INDEX <index name> ON <table name>
```

To use this statement, you must specify the DROP INDEX keywords, the name of the index, and the name of the table in the ON clause, as shown in the following example:

```
DROP INDEX unique_1 ON Orders;
```

This DROP INDEX statement removes the unique_1 index from the Orders table, just as the ALTER TABLE statement does previously.

Retrieving Information About Database Objects

Up till this point in the chapter, you've been creating, modifying, and deleting objects in MySQL. MySQL, however, also provides methods that allow you to view information about those objects. In this section, you learn about a number of statements that display information about databases and their tables. The statements can be divided into two broad categories: SHOW statements and DESCRIBE statements.

Using SHOW Statements

The SHOW statements in MySQL display a variety of information about databases and their tables. The database-related SHOW statements include the SHOW CREATE DATABASE and SHOW DATABASES statement. The table-related SHOW statements include SHOW COLUMNS, SHOW CREATE TABLE, SHOW INDEX, and SHOW TABLES. When using the table-related statements, you should be working in the context of the database, unless you specify the database name as part of the SHOW statement.

Using Database-Related SHOW Statements

The SHOW CREATE DATABASE statement allows you to view the database definition for a specific database. The following syntax shows how to create a SHOW CREATE DATABASE statement:

```
SHOW CREATE DATABASE <database name>
```

As you can see, you need to specify only the SHOW CREATE DATABASE keywords and the name of the database, as shown in the following example:

```
SHOW CREATE DATABASE mysql;
```

In this case, the statement retrieves information about the mysql database definition. When you execute the statement, you should receive results similar to the following:

```
+----------+-----------------------------------------------------------------+
| Database | Create Database                                                 |
+----------+-----------------------------------------------------------------+
| mysql    | CREATE DATABASE `mysql` /*!40100 DEFAULT CHARACTER SET latin1 */ |
+----------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

The entire results cannot be displayed here, because the row is too long. The actual results that you see depend on your system, but the basic information is the same.

The next statement to examine is the SHOW DATABASE statement. The statement lists the MySQL databases that exist in your system. The following syntax illustrates the SHOW DATABASE statement:

```
SHOW DATABASES [LIKE '<value>']
```

Notice that the statement includes an optional LIKE clause. The clause lets you specify a value for database names. MySQL returns only names of databases that match that value. You usually use the LIKE clause in conjunction with a wildcard to return names that are similar to the specified value. (In SQL, the percent [%] character serves as a wildcard in much the same way as the asterisk [*] character serves as a wildcard in other applications.) For example, the following SHOW DATABASE returns only those databases whose name begins with "my:"

```
SHOW DATABASES LIKE 'my%';
```

When you execute this statement, you should receive results similar to the following:

```
+----------------+
| Database (my%) |
+----------------+
| mysql          |
+----------------+
1 row in set (0.00 sec)
```

As you can see, only databases that begin with "my" are returned. In this case, only the mysql database is displayed.

Using Table-Related SHOW Statements

The `SHOW COLUMNS` statement lists the columns in a table, along with information about the columns. The following syntax illustrates the `SHOW COLUMNS` statement:

```
SHOW [FULL] COLUMNS FROM <table name> [FROM <database name>] [LIKE '<value>']
```

To use this statement, you must specify the `SHOW COLUMNS FROM` keywords, the name of the table, and optionally the name of the database, if you're not working in the context of that database. To show more complete information about each column, you should also use the `FULL` keyword. In addition, you can use the `LIKE` clause to limit the values returned, as shown in the following example:

```
SHOW COLUMNS FROM user FROM mysql LIKE '%priv';
```

This `SHOW COLUMNS` statement returns column information from the user table in the mysql database. The `LIKE` clause limits the columns returned to those ending with "priv." (The % wildcard indicates that the value can begin with any characters.) The `SHOW COLUMNS` statement shown here should produce results similar to the following:

```
+-----------------------+---------------+------+-----+---------+-------+
| Field                 | Type          | Null | Key | Default | Extra |
+-----------------------+---------------+------+-----+---------+-------+
| Select_priv           | enum('N','Y') |      |     | N       |       |
| Insert_priv           | enum('N','Y') |      |     | N       |       |
| Update_priv           | enum('N','Y') |      |     | N       |       |
| Delete_priv           | enum('N','Y') |      |     | N       |       |
| Create_priv           | enum('N','Y') |      |     | N       |       |
| Drop_priv             | enum('N','Y') |      |     | N       |       |
| Reload_priv           | enum('N','Y') |      |     | N       |       |
| Shutdown_priv         | enum('N','Y') |      |     | N       |       |
| Process_priv          | enum('N','Y') |      |     | N       |       |
| File_priv             | enum('N','Y') |      |     | N       |       |
| Grant_priv            | enum('N','Y') |      |     | N       |       |
| References_priv       | enum('N','Y') |      |     | N       |       |
| Index_priv            | enum('N','Y') |      |     | N       |       |
| Alter_priv            | enum('N','Y') |      |     | N       |       |
| Show_db_priv          | enum('N','Y') |      |     | N       |       |
| Super_priv            | enum('N','Y') |      |     | N       |       |
| Create_tmp_table_priv | enum('N','Y') |      |     | N       |       |
| Lock_tables_priv      | enum('N','Y') |      |     | N       |       |
| Execute_priv          | enum('N','Y') |      |     | N       |       |
| Repl_slave_priv       | enum('N','Y') |      |     | N       |       |
| Repl_client_priv      | enum('N','Y') |      |     | N       |       |
+-----------------------+---------------+------+-----+---------+-------+
21 rows in set (0.00 sec)
```

Notice that each column name ends in "priv." Also notice that the results include details about each column. You can also retrieve information about a table by using a `SHOW CREATE TABLE` statement, which displays the table definition. The following syntax shows how to create a `SHOW CREATE TABLE` statement:

```
SHOW CREATE TABLE <table name>
```

In this statement, you need to specify the `SHOW CREATE TABLE` keywords, along with the name of the table, as shown in the following example:

```
SHOW CREATE TABLE func;
```

This `SHOW CREATE TABLE` statement returns the table definition for the func table. When you execute this statement, you should receive results similar to the following:

```
+-------+-----------------------------------------------------------------------
| Table | Create Table
+-------+-----------------------------------------------------------------------
| func  | CREATE TABLE `func` (`name` char(64) character set latin1 collate latin1_
+-------+-----------------------------------------------------------------------
1 row in set (0.02 sec)
```

The entire results could not fit on the screen because the row is too long. The data that you see depends on your system.

The next statement displays a list of indexes in a table. The `SHOW INDEX` statement is shown in the following syntax:

```
SHOW INDEX FROM <table name> [FROM <database name>]
```

The only required elements of this statement are the `SHOW INDEX FROM` keywords and the name of the table. You can also use the `FROM` clause to specify the name of the database, which you would do if you're not working in the context of that database. For example, the following `SHOW INDEX` statement displays information about the indexes in the user table in the mysql database:

```
SHOW INDEX FROM user FROM mysql;
```

When you execute this statement, you should receive results similar to the following:

```
+-------+------------+----------+--------------+-------------+-----------+---------
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinal
+-------+------------+----------+--------------+-------------+-----------+---------
| user  |          0 | PRIMARY  |            1 | Host        | A         |        N
| user  |          0 | PRIMARY  |            2 | User        | A         |
+-------+------------+----------+--------------+-------------+-----------+---------
2 rows in set (0.00 sec)
```

Once again, the entire results are not displayed here because the rows are too long. The exact results that you see on your system vary; however, the basic information should be the same.

The next statement is the `SHOW TABLES` statement, which displays a list of tables in the current database or a specified database. The syntax for the statement is as follows:

```
SHOW TABLES [FROM <database name>] [LIKE '<value>']
```

As you can see, the only elements that you need to specify are the SHOW TABLES keywords. You can also specify the database in the FROM clause, and you can specify a value in the LIKE clause, as shown in the following example:

```
SHOW TABLES FROM mysql LIKE 'help%';
```

In this SHOW TABLES statement, you display all tables in the mysql database that begin with "help," as shown in following results:

```
+-------------------------+
| Tables_in_mysql (help%) |
+-------------------------+
| help_category           |
| help_keyword            |
| help_relation           |
| help_topic              |
+-------------------------+
4 rows in set (0.00 sec)
```

As you can see, the list includes only tables that begin with "help."

Using DESCRIBE Statements

Another statement useful for viewing information about tables is the DESCRIBE statement. The following syntax describes how to define a DESCRIBE statement:

```
DESCRIBE <table name> [<column name> | '<value>']
```

The only required elements of the DESCRIBE statement are the DESCRIBE keyword and the name of the table. You can also specify a column name or a value used to return columns with names similar to the value, in which case you would use a wildcard. The following example shows a DESCRIBE statement that returns information about all columns in the user table that end with "priv":

```
DESCRIBE user '%priv';
```

If you execute this statement, you should receive results similar to the following:

```
+-----------------------+---------------+------+-----+---------+-------+
| Field                 | Type          | Null | Key | Default | Extra |
+-----------------------+---------------+------+-----+---------+-------+
| Select_priv           | enum('N','Y') |      |     | N       |       |
| Insert_priv           | enum('N','Y') |      |     | N       |       |
| Update_priv           | enum('N','Y') |      |     | N       |       |
| Delete_priv           | enum('N','Y') |      |     | N       |       |
| Create_priv           | enum('N','Y') |      |     | N       |       |
| Drop_priv             | enum('N','Y') |      |     | N       |       |
| Reload_priv           | enum('N','Y') |      |     | N       |       |
| Shutdown_priv         | enum('N','Y') |      |     | N       |       |
| Process_priv          | enum('N','Y') |      |     | N       |       |
| File_priv             | enum('N','Y') |      |     | N       |       |
| Grant_priv            | enum('N','Y') |      |     | N       |       |
```

```
| References_priv       | enum('N','Y') |      |     | N       |       |
| Index_priv            | enum('N','Y') |      |     | N       |       |
| Alter_priv            | enum('N','Y') |      |     | N       |       |
| Show_db_priv          | enum('N','Y') |      |     | N       |       |
| Super_priv            | enum('N','Y') |      |     | N       |       |
| Create_tmp_table_priv | enum('N','Y') |      |     | N       |       |
| Lock_tables_priv      | enum('N','Y') |      |     | N       |       |
| Execute_priv          | enum('N','Y') |      |     | N       |       |
| Repl_slave_priv       | enum('N','Y') |      |     | N       |       |
| Repl_client_priv      | enum('N','Y') |      |     | N       |       |
+-----------------------+---------------+------+-----+---------+-------+
21 rows in set (0.00 sec)
```

Notice that only columns that end in "priv" are displayed. The `DESCRIBE` statement is a handy way to view the information that you need quickly.

In this exercise, you try out some of the `SHOW` and `DESCRIBE` statements that you learned about in this section of the chapter.

Try It Out Displaying Database Information

The following steps lead you through a series of statements:

1. Open the mysql client utility, type the following command, and press Enter:

```
SHOW DATABASES;
```

You should receive results similar to the following:

```
+------------+
| Database   |
+------------+
| dvdrentals |
| mysql      |
| test       |
+------------+
3 rows in set (0.00 sec)
```

At the very least, you should see the two databases that are installed by default — mysql and test — and the DVDRentals database, which you created earlier in the chapter.

2. Next, you view the `CREATE DATABASE` statement for the DVDRentals database. To view the database definition, type the following `SHOW CREATE DATABASE` statement at the mysql command prompt, and then press Enter:

```
SHOW CREATE DATABASE DVDRentals;
```

You should receive results similar to the following:

```
+------------+-----------------------------------------------------------------
| Database   | Create Database
+------------+-----------------------------------------------------------------
| dvdrentals | CREATE DATABASE `dvdrentals` /*!40100 DEFAULT CHARACTER SET latin1 *
+------------+-----------------------------------------------------------------
1 row in set (0.01 sec)
```

Because the row is so long, only a part of the results are displayed here. The amount of data that is displayed on your system and the way that it is displayed vary from system to system. As a result, you might have to scroll to the right or up and down to view all the results.

3. Now you will switch to the DVDRentals database. Type the following command at the mysql command prompt, and press Enter:

```
use DVDRentals
```

You should receive a message indicating that you switched to the DVDRentals database.

4. To display a list of the tables in the DVDRentals database, type the following command at the mysql command prompt, and then press Enter:

```
SHOW TABLES;
```

You should see results similar to the following:

```
+----------------------+
| Tables_in_dvdrentals |
+----------------------+
| customers            |
| dvdparticipant       |
| dvds                 |
| employees            |
| formats              |
| movietypes           |
| orders               |
| participants         |
| ratings              |
| roles                |
| status               |
| studios              |
| transactions         |
+----------------------+
13 rows in set (0.00 sec)
```

All the tables that you created earlier in the chapter should be displayed. As you can see, there are 13 tables in all.

5. Next, view the table definition for the Orders table. Type the following `SHOW CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
SHOW CREATE TABLE Orders;
```

You should receive results similar to the following:

```
+--------+-------------------------------------------------------------------------
| Table  | Create Table
+--------+-------------------------------------------------------------------------
| Orders | CREATE TABLE `orders` (`OrderID` int(11) NOT NULL auto_increment, `CustI
+--------+-------------------------------------------------------------------------
1 row in set (0.01 sec)
```

As before, only part of the results can be displayed here. The amount of data displayed and the way it will be displayed vary from system to system.

6. Next, display the columns that are in the Transactions table. Type the following command at the mysql command prompt, and then press Enter:

```
SHOW COLUMNS FROM Transactions;
```

You should receive results similar to the following:

```
+---------+-------------+------+-----+------------+----------------+
| Field   | Type        | Null | Key | Default    | Extra          |
+---------+-------------+------+-----+------------+----------------+
| TransID | int(11)     |      | PRI | NULL       | auto_increment |
| OrderID | int(11)     |      | MUL | 0          |                |
| DVDID   | smallint(6) |      | MUL | 0          |                |
| DateOut | date        |      |     | 0000-00-00 |                |
| DateDue | date        |      |     | 0000-00-00 |                |
| DateIn  | date        |      |     | 0000-00-00 |                |
+---------+-------------+------+-----+------------+----------------+
6 rows in set (0.00 sec)
```

Notice that each column is listed, along with the data type, the column's nullability, and additional column settings.

7. Another way to view information about a table is to use a `DESCRIBE` statement. Type the following command at a mysql command prompt, and then press Enter:

```
DESCRIBE DVDParticipant;
```

You should receive results similar to the following:

```
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| DVDID  | smallint(6) |      | PRI | 0       |       |
| PartID | smallint(6) |      | PRI | 0       |       |
| RoleID | varchar(4)  |      | PRI |         |       |
+--------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

As with the `SHOW COLUMNS` statement, you see a list of columns, along with information about each column.

8. The final step is to view the indexes that have been created on a table. Type the following `SHOW INDEX` statement at the mysql command prompt, and then press Enter:

```
SHOW INDEX FROM DVDs;
```

You should see results similar to the following:

```
+-------+------------+----------+--------------+-------------+-----------+---------
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinal
+-------+------------+----------+--------------+-------------+-----------+---------
| DVDs  |          0 | PRIMARY  |            1 | DVDID       | A         |
| DVDs  |          1 | MTypeID  |            1 | MTypeID     | A         |
| DVDs  |          1 | StudID   |            1 | StudID      | A         |
| DVDs  |          1 | RatingID |            1 | RatingID    | A         |
| DVDs  |          1 | FormID   |            1 | FormID      | A         |
| DVDs  |          1 | StatID   |            1 | StatID      | A         |
+-------+------------+----------+--------------+-------------+-----------+---------
6 rows in set (0.00 sec)
```

Notice that the primary key and foreign keys are listed as indexes. As you learned earlier in the chapter, there are a number of different types of indexes, including primary keys and foreign keys.

How It Works

In this exercise, you executed a number of `SHOW` statements and one `DESCRIBE` statement to view information about the databases that exist in your server, to view the DVDRentals database definition, and to view information about different tables in the DVDRentals database. As you have seen, these statements can be very useful when trying to find information about an existing database and the tables in that database.

As you work your way through this book, you might find that you use these statements often. In that case, you should refer to this chapter as necessary to reference these commands.

Summary

In this chapter, you learned how to create, modify, and remove databases, tables, and indexes from your system. The chapter provided the syntax of the various statements necessary to perform these tasks, explained how to use the syntax to create SQL statements, and provided examples that demonstrated how to implement the various statements. In addition, you created the tables necessary to support the DVDRentals database. The tables were based on the data model that you created in Chapter 4. The chapter also explained how to use `SHOW` and `DESCRIBE` statements to view information about your database. Specifically, the chapter provided you with the information you need to perform the following tasks:

- Create a database definition that specifies the character set and collation for that database.
- Modify the character set and collation associated with a database.
- Delete a database.
- Create a table definition that includes column definitions, primary keys, foreign keys, and indexes.
- Alter table definitions, including adding, modifying, and removing columns, primary keys, foreign keys, and indexes.
- Remove tables from a database.
- Generate SQL statements that retrieve information about the databases and tables in your system.

Once you know how to create a database, add tables to the database, and configure the elements in the tables, you can create the tables that you need to support your databases. In these tables you can insert, modify, and delete data. In the next chapter, you learn how to manage data in your MySQL database. From there, you learn how to retrieve data, import and export data, and manage transactions. You can even create applications that can access the data from within the application languages.

Exercises

The following exercises help you build on the information you learned in this chapter. To view the answers, see Appendix A.

1. You are creating a database named NewDB. The database uses the server's default character set but uses the `latin1_general_ci` collation. What SQL statement should you use to create the database?
2. You are creating a table in a MySQL database. The table is named Bikes and includes two columns: BikeID and BikeName. The BikeID column must uniquely identify each row in the table, and values must be automatically assigned to each row. In addition, the table should never contain more than 200 models of bikes. The BikeName column must include a descriptive name for each model of bike. The names vary in length but should never exceed 40 characters. In addition, the table never participates in a transaction or foreign key relationship. What SQL statement should you use to create the table?
3. You plan to add a unique index to a table that was defined with the following CREATE TABLE statement:

```
CREATE TABLE ModelTrains
(
ModelID SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
ModelName VARCHAR(40) NOT NULL
);
```

   The index should be named un_1 and should be configured on the ModelName column. What `ALTER TABLE` statement should you use to add the index?
4. You now want to drop the un_1 unique index from the ModelTrains table. What `ALTER TABLE` statement should you use to remove the unique index?
5. What SQL statement should you use if you want to view a list of tables in the current database?

6

Manipulating Data in a MySQL Database

At the heart of every RDBMS is the data stored in that system. The RDBMS is configured and the database is designed for the sole purpose of managing data storage, access, and integrity. Ultimately, the purpose of any RDBMS is to manage data. For this reason, this chapter and the following five chapters focus exclusively on data access and manipulation. You learn how to add and modify data, retrieve data, and use advanced techniques to carry out these operations.

To start you off in the direction of data management, this chapter introduces you to the SQL statements available in MySQL to manipulate data. The statements provide numerous options for effectively inserting data into a database, updating that data, and deleting it once it is no longer useful. By the end of the chapter, you'll be able to populate your database tables according to the restrictions placed on those tables, and you'll be able to access those tables to perform the necessary updates and deletions. Specifically, the chapter covers the following topics:

- How to use `INSERT` and `REPLACE` statements to add data to tables in a MySQL database
- How to use `UPDATE` statements to modify data in tables in a MySQL database
- How to use `DELETE` and `TRUNCATE` statements to delete data from tables in a MySQL database

Inserting Data in a MySQL Database

Before you can do anything with the data in a database, the data must exist. For this reason, the first SQL statements that you need to learn are those that insert data in a MySQL database. When you add data to a database, you're actually adding it to the individual tables in that database. Because of this, you must remain aware of how tables relate to each other and whether foreign keys have been defined on any of their columns. For example, in a default configuration of a foreign key, you cannot add a row to a child table if the referencing column of the inserted row contains a value that does not exist in the referenced column of the parent table. If you try to do so, you receive an error. (For more information about foreign keys, refer to Chapter 5.)

MySQL supports two types of statements that insert values in a table: INSERT and REPLACE. Both statements allow you to add data directly to the tables in your MySQL database. MySQL also supports other methods for inserting data in a table, such as copying or importing data. These methods are discussed in Chapter 11.

Using an INSERT Statement to Add Data

An INSERT statement is the most common method used to directly insert data in a table. The statement provides a great deal of flexibility in defining the values for individual rows or for multiple rows. The following syntax defines each element that makes up the INSERT statement:

```
<insert statement>::=
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]
{<values option> | <set option> | <select option>}

<values option>::=
<table name> [(<column name> [{, <column name>}...])]
VALUES ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])
   [{, ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])}...]

<set option>::=
<table name>
SET <column name>={<expression> | DEFAULT}
   [{, <column name>={<expression> | DEFAULT}}...]

<select option>::=
<table name> [(<column name> [{, <column name>}...])]
<select statement>
```

You can use the INSERT statement to add data to any table in a MySQL database. When you add data, you must do so on a row-by-row basis, and you must insert exactly one value per column. If you specify fewer values than there are columns, default or null values are inserted for the unspecified values.

Now take a look at the first line of the INSERT statement syntax:

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]
```

As you can see, the first line includes the required INSERT keyword and a number of options. The first of these options are LOW_PRIORITY and DELAYED. You can include either one of these options, but you cannot include both. If you specify LOW_PRIORITY, the statement is not executed until no other client connections are accessing the same table that the INSERT statement is accessing. This can result in a long delay while you're waiting for the statement to execute. During that time, you cannot take any other actions. If you specify DELAYED, the execution is also delayed, but you can continue to take other actions while the INSERT statement is in queue. The LOW_PRIORITY and DELAYED options can be used only for inserts against MyISAM and ISAM tables.

The next option that you can specify in the INSERT clause is IGNORE. This option applies primarily to INSERT statements that add multiple rows to a table. If you specify IGNORE, inserted rows are ignored if they contain a value that duplicates a primary key or unique index value. The INSERT statement continues to insert the remaining rows. If you do not specify IGNORE, the supplicated values abort the insert process.

The final option in the INSERT clause is the INTO keyword. The keyword has no impact on how the INSERT statement processes, although it is used quite often as a way to provide a sort of roadmap to the statement by indicating the table that is the target of the inserted rows.

Moving on to the next line of syntax in the INSERT statement, you can see that there are three options from which to choose:

```
{<values option> | <set option> | <select option>}
```

Each option refers to a clause in the INSERT statement that helps to define the values to insert in the target table. The remaining part of the syntax defines each INSERT alternative. As you can see from the syntax, a number of elements are common to all three statement options. One of these elements — the *<expression>* placeholder — is of particular importance to the INSERT statement. An *expression* is a type of formula that helps define the value to insert in a column. In many cases, an expression is nothing more than a *literal value,* which is another way of referring to a value exactly as it will be inserted in a table. In addition to literal values, expressions can also include column names, operators, and functions. An *operator* is a symbol that represents a particular sort of action that should be taken, such as comparing values or adding values together. For example, the plus (+) sign is an arithmetic operator that adds two values together. A *function,* on the other hand, is an object that carries out a predefined task. For example, you can use a function to specify the current date.

You can use expressions in your INSERT statement to help define the data to insert in a particular column. Because operators and functions are not discussed until later chapters (Chapters 8 and 9, respectively), the inclusion of expressions in the example INSERT statements in this chapter, except for the most basic expressions, is minimal. Later in the book, when you have a better comprehension of how to use functions and operators, you'll have a better sense of how to include complex expressions in your INSERT statements.

For now, the discussion moves on to the three INSERT alternatives included in the syntax. Each alternative provides a method for defining the values that are inserted in the target table. The *<values option>* alternative defines the values in the VALUES clause, the *<set option>* defines the values in the SET clause, and the *<select option>* defines the values in a SELECT statement, which is embedded in the INSERT statement. Because SELECT statements are not discussed until later in the book, the *<select option>* alternative is not discussed until Chapter 11. The following two sections discuss how to create INSERT statements using the other two alternatives.

Using the <*values option*> Alternative of the INSERT Statement

The *<values option>* alternative of the INSERT statement allows you to add one or more rows to a table, as shown in the following syntax:

```
<values option>::=
<table name> [(<column name> [{, <column name>}...])]
VALUES ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])
   [{, ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])}...]
```

As the syntax shows, you must provide a table name and a VALUES clause. You also have the option of specifying one or more columns after the table name. If you specify column names, they must be enclosed in parentheses and separated by commas.

Once you specify the table and optional column names, you must specify a VALUES clause. The clause must include at least one value, which is represented by the *<expression>* placeholder or the DEFAULT keyword. If you include column names after the table name, the VALUES clause must include a value for each column, in the order that the columns are listed. If you did not specify column names, you must provide a value for every column in the table, in the order that the columns are defined in the table.

> *If you're uncertain of the names of columns or the order in which they are defined in a table, you can use a* DESCRIBE *statement to view a list of the columns names. See Chapter 5 for more information about the* DESCRIBE *statement.*

All values must be enclosed in parentheses. If there are multiple values, they must be separated by commas. For columns configured with the AUTO_INCREMENT option or a TIMESTAMP data type, you can specify NULL rather than an actual value, or omit the column and value altogether. Doing so inserts the correct value into those columns. In addition, you can use the DEFAULT keyword in any place that you want the default value for that column inserted in the table. (Be sure to refer back to Chapter 5 for a complete description on how MySQL handles default values.)

Now that you have a basic overview of the syntax, take a look at a few examples of how to create an INSERT statement. The examples are based on the following table definition:

```
CREATE TABLE CDs
(
   CDID SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
   CDName VARCHAR(50) NOT NULL,
   Copyright YEAR,
   NumberDisks TINYINT UNSIGNED NOT NULL DEFAULT 1,
   NumberInStock TINYINT UNSIGNED,
   NumberOnReserve TINYINT UNSIGNED NOT NULL,
   NumberAvailable TINYINT UNSIGNED NOT NULL,
   CDType VARCHAR(20),
   RowAdded TIMESTAMP
);
```

You should be well familiar with all the elements in the table definition. If you have any questions about any component, refer back to Chapter 5 for an explanation. Nothing has been used here that you have not already seen.

> *If you want to create the CDs table (or any of the example tables in this chapter) and try out the example SQL statements, you should use the test database or you should create a database for specifically for this purpose.*

The table in this definition, which is named CDs, stores information about compact disks. Suppose that you want to use an INSERT statement to add information about a CD named *Ain't Ever Satisfied: The Steve Earle Collection*. You can set up your statement in a couple of ways. The first is to specify a value for each column, without specifying the name of the columns, as shown in the following INSERT statement:

```
INSERT INTO CDs
VALUES (NULL, 'Ain\'t Ever Satisfied: The Steve Earle Collection',
   1996, 2, 10, 3, NumberInStock-NumberOnReserve, 'Country', NULL);
```

In this statement, the first line contains the mandatory keyword INSERT, the optional keyword INTO, and the name of the table (CDs). The VALUES clause includes a value for each column, entered in the order in which the columns appear in the table definition. The values are enclosed in parentheses and separated with commas.

The first specified value is NULL. The value is used for the CDID column, which is configured with the AUTO_INCREMENT option and is the primary key. By specifying NULL, the next incremented value is automatically inserted in that column when you add this row to the table. Because this is the first row added to the table, a value of 1 is inserted in the CDID column.

The next value in the VALUES clause corresponds to the CDName column. Because this value is a string, it is enclosed in parentheses. In addition, the backslash precedes the apostrophe. The backslash is used in a string value to notify MySQL that the following character is a literal value and should not be interpreted as the ending quote of the string. The backslash is useful for any characters that could be misinterpreted when executing a statement that contains a string value.

> *You can also use the backslash to specify other literal values, such as double quotes (\"), a backslash (\\), a percentage sign (\%) or an underscore (_).*

The next four values specified in the VALUES clause are date and numerical data that correspond with the columns in the table definition (Copyright = 1996, NumberDisks = 2, NumberInStock = 10, and NumberOnReserve = 3).

The value specified for the NumberAvailable column (NumberInStock-NumberOnReserve) is an expression that uses two column names and the minus (-) arithmetic operator to subtract the value in the NumberOnReserve column from the NumberInStock column to arrive at a total of the number of CDs available for sale. In this case, the total is 7.

The next value specified in the VALUES clause is Country, which is inserted in the CDType column. The final value is NULL, which is used for the RowAdded column. The column is configured as a TIMESTAMP column, which means that the current time and date are inserted automatically in that column.

Another way that you can insert the current date in the table is to use the NOW() function, rather than NULL. The NOW() function can be used in SQL statements to return a value that is equivalent to the current date and time. When used in an INSERT statement, that value can be added to a time/date column. If you want to retrieve only the current date, and not the time, you can use the CURDATE() function. If you want to retrieve only the current time, and not the current date, you can use the CURTIME() function.

In addition to the functions mentioned here, you can use other functions to insert data into a table. Chapter 9 describes many of the functions available in MySQL and how you can use those functions in your SQL statements.

The next example INSERT statement also adds a row to the CDs table. In this statement, the columns names are specified and only the values for those columns are included, as the following statement demonstrates:

```
INSERT LOW_PRIORITY INTO CDs (CDName, Copyright, NumberDisks,
   NumberInStock, NumberOnReserve, NumberAvailable, CDType)
VALUES ('After the Rain: The Soft Sounds of Erik Satie',
   1995, DEFAULT, 13, 2, NumberInStock - NumberOnReserve, 'Classical');
```

In this statement, the CDID column and the RowAdded column are not specified. Because CDID is an AUTO_INCREMENT column, an incremented value is automatically inserted in that column. Because the RowAdded column is a TIMESTAMP column, the current date and time are inserted in the column. Whenever a column is not specified in an INSERT statement, the default value for that column is inserted in the column.

The values that are specified in this INSERT statement are similar to the previous example except for one difference. For the NumberDisks column, DEFAULT is used. This indicates that the default value should be inserted in that column. In this case, that value is 1. Otherwise, the values are listed in a matter similar to what you saw previously. There is one other difference between the two statements, however. The last example includes the LOW_PRIORITY option in the INSERT clause. As a result, this statement is not processed and the client is put on hold until all other client connections have completed accessing the target table.

To demonstrate the types of values that are inserted in the CDID column and the NumberDisks column of the CDs table by the last two example, you can use the following SELECT statement to retrieve data from the CDID, CDName, and NumberDisks of the columns the CDs table:

```
SELECT CDID, CDName, NumberDisks
FROM CDs;
```

The SELECT statement returns results similar to the following

```
+------+-------------------------------------------------+-------------+
| CDID | CDName                                          | NumberDisks |
+------+-------------------------------------------------+-------------+
|    1 | Ain't Ever Satisfied: The Steve Earle Collection |           2 |
|    2 | After the Rain: The Soft Sounds of Erik Satie    |           1 |
+------+-------------------------------------------------+-------------+
2 rows in set (0.00 sec)
```

As you can see, incremented values have been inserted in the CDID column, and a value of 1 has been inserted in the NumberDisks column of the second row.

The next example INSERT statement is much simpler than the rest. It specifies the value for only the CDName column, as shown in the following statement:

```
INSERT INTO CDs (CDName)
VALUES ('Blue');
```

Because the statement specifies only one value, default values are inserted for all other columns. This approach is fine for the CDID, NumberDisks, and RowAdded columns, but it could be problematic for the other columns, unless those are the values you want. For example, because null values are permitted in the Copyright column, NULL was inserted. The NumberInStock and CDType columns also permit null values, so NULL was inserted in those two as well. On the other hand, the NumberOnReserve and NumberAvailable columns do not permit null values, so 0 was inserted into these columns. As a result, whenever you're inserting rows into a table, you must be aware of what the default value is in each column, and you must be sure to specify all the necessary values.

The last example of an INSERT statement that uses the *<values option>* alternative inserts values into multiple rows. The following INSERT statement includes the name of the columns in the INSERT clause and then specifies the values for those columns for three rows:

```
INSERT INTO CDs (CDName, Copyright, NumberDisks,
   NumberInStock, NumberOnReserve, NumberAvailable, CDType)
VALUES ('Mule Variations', 1999, 1, 9, 0,
   NumberInStock-NumberOnReserve, 'Blues'),
('The Bonnie Raitt Collection', 1990, 1, 14, 2,
   NumberInStock-NumberOnReserve, 'Popular'),
('Short Sharp Shocked', 1988, 1, 6, 1,
   NumberInStock-NumberOnReserve, 'Folk-Rock');
```

As the statement demonstrates, when using one INSERT statement to insert values in multiple rows, you must enclose the values for each row in their own set of parentheses, and you must separate the sets of values with a comma. By using this method, you can insert as many rows as necessary in your table, without having to generate multiple INSERT statements.

Using the <*values option*> Alternative to Insert Data in the DVDRentals Database

In Chapter 5, you created the tables for the DVDRentals database. In this chapter you populate those tables with data. The following three Try It Out sections walk you through the steps necessary to add the initial data to each table. Because some dependencies exist between tables — foreign keys have been defined on numerous columns — you must add data to the referenced parent tables before adding data to the referencing child tables. To facilitate this process, first add data to the lookup tables, then to the people tables, and finally those tables configured with foreign keys.

Try It Out Inserting Data in the Lookup Tables

The lookup tables contain the data that generally changes more infrequently than other data. To insert data in these tables, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To insert a record int the Formats table, type the following INSERT statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Formats
VALUES ('f1', 'Widescreen');
```

   You should receive a response saying that your statement executed successfully, affecting one row.

3. To insert the next record in the Formats table, type the following INSERT statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Formats (FormID, FormDescrip)
VALUES ('f2', 'Fullscreen');
```

   You should receive a response saying that your statement executed successfully, affecting one row.

4. To insert records in the Roles table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Roles
VALUES ('r101', 'Actor'),
('r102', 'Director'),
('r103', 'Producer'),
('r104', 'Executive Producer'),
('r105', 'Co-Producer'),
('r106', 'Assistant Producer'),
('r107', 'Screenwriter'),
('r108', 'Composer');
```

You should receive a response saying that your statement executed successfully, affecting eight rows.

5. To insert records in the MovieTypes table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO MovieTypes
VALUES ('mt10', 'Action'),
('mt11', 'Drama'),
('mt12', 'Comedy'),
('mt13', 'Romantic Comedy'),
('mt14', 'Science Fiction/Fantasy'),
('mt15', 'Documentary'),
('mt16', 'Musical');
```

You should receive a response saying that your statement executed successfully, affecting seven rows.

6. To insert records in the Studios table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Studios
VALUES ('s101', 'Universal Studios'),
('s102', 'Warner Brothers'),
('s103', 'Time Warner'),
('s104', 'Columbia Pictures'),
('s105', 'Paramount Pictures'),
('s106', 'Twentieth Century Fox'),
('s107', 'Merchant Ivory Production');
```

You should receive a response saying that your statement executed successfully, affecting seven rows.

7. To insert records in the Ratings table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Ratings
VALUES ('NR', 'Not rated'),
('G', 'General audiences'),
('PG', 'Parental guidance suggested'),
('PG13', 'Parents strongly cautioned'),
('R', 'Under 17 requires adult'),
('X', 'No one 17 and under');
```

You should receive a response saying that your statement executed successfully, affecting six rows.

8. To insert records in the Status table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Status
VALUES ('s1', 'Checked out'),
('s2', 'Available'),
('s3', 'Damaged'),
('s4', 'Lost');
```

You should receive a response saying that your statement executed successfully, affecting four rows.

How It Works

In this exercise, you added data to the five lookup tables in the DVDRentals database. The first table that you populated was the Formats table. You used the following `INSERT` statement to add a single row to the table:

```
INSERT INTO Formats
VALUES ('f1', 'Widescreen');
```

Because you did not specify the columns in this statement, you had to specify values for all columns in the table. The table contains only two columns, and the primary key values are not generated automatically, so you were required to specify all the values anyway. As a result, specifying the column names wasn't necessary. In the next step, you did specify the column names, as shown in the following statement:

```
INSERT INTO Formats (FormID, FormDescrip)
VALUES ('f2', 'Fullscreen');
```

As you can see, the results were the same as in the previous steps. Specifying the column names in this case required extra effort but provided no added benefit.

In the remaining steps, you used individual `INSERT` statements to add multiple rows to each table. For example, you used the following `INSERT` statement to add data to the Roles table:

```
INSERT INTO Roles
VALUES ('r101', 'Actor'),
('r102', 'Director'),
('r103', 'Producer'),
('r104', 'Executive Producer'),
('r105', 'Co-Producer'),
('r106', 'Assistant Producer'),
('r107', 'Screenwriter'),
('r108', 'Composer');
```

Because the table contains only two columns and both values are provided for each row, you are not required to specify the columns' names. You do need to enclose the values for each row in parentheses and separate each set of values by commas.

The values for the remaining tables were inserted by using the same format as the `INSERT` statement used for the Roles table. In each case, the columns were not specified, multiple rows were inserted, and both values were provided for each row.

Once you insert the data into the lookup tables, you're ready to add data to the tables that contain information about the people whose participation is recorded in the database, such as employees, customers, and those who make the movies.

Try It Out Inserting Data in the People Tables

To insert data in these tables, follow these steps:

1. If it's not already open, open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To insert records in the Participants table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Participants (PartFN, PartMN, PartLN)
VALUES ('Sydney', NULL, 'Pollack'),
('Robert', NULL, 'Redford'),
('Meryl', NULL, 'Streep'),
('John', NULL, 'Barry'),
('Henry', NULL, 'Buck'),
('Humphrey', NULL, 'Bogart'),
('Danny', NULL, 'Kaye'),
('Rosemary', NULL, 'Clooney'),
('Irving', NULL, 'Berlin'),
('Michael', NULL, 'Curtiz'),
('Bing', NULL, 'Crosby');
```

   You should receive a response saying that your statement executed successfully, affecting 11 rows.

3. To insert records in the Employees table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Employees (EmpFN, EmpMN, EmpLN)
VALUES ('John', 'P.', 'Smith'),
('Robert', NULL, 'Schroader'),
('Mary', 'Marie', 'Michaels'),
('John', NULL, 'Laguci'),
('Rita', 'C.', 'Carter'),
('George', NULL, 'Brooks');
```

   You should receive a response saying that your statement executed successfully, affecting six rows.

4. To insert records in the Customers table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Customers (CustFN, CustMN, CustLN)
VALUES ('Ralph', 'Frederick', 'Johnson'),
('Hubert', 'T.', 'Weatherby'),
('Anne', NULL, 'Thomas'),
('Mona', 'J.', 'Cavenaugh'),
('Peter', NULL, 'Taylor'),
('Ginger', 'Meagan', 'Delaney');
```

You should receive a response saying that your statement executed successfully, affecting six rows.

How It Works

In this exercise, you used three `INSERT` statements to insert data in the three people tables (one statement per table). Each statement was identical in structure. The only differences were in the values defined and the number of rows inserted. For example, you used the following statement to insert data in the Participants table:

```
INSERT INTO Participants (PartFN, PartMN, PartLN)
VALUES ('Sydney', NULL, 'Pollack'),
('Robert', NULL, 'Redford'),
('Meryl', NULL, 'Streep'),
('John', NULL, 'Barry'),
('Henry', NULL, 'Buck'),
('Humphrey', NULL, 'Bogart'),
('Danny', NULL, 'Kaye'),
('Rosemary', NULL, 'Clooney'),
('Irving', NULL, 'Berlin'),
('Michael', NULL, 'Curtiz'),
('Bing', NULL, 'Crosby');
```

As you can see, the `INSERT` clause includes the `INTO` option, which has no effect on the statement, and the name of the columns. Because the PartID column is configured with the `AUTO_INCREMENT` option, you don't need to include it here. The new value is automatically inserted in the column. Because these are the first rows to be added to the table, a value of 1 is added to the PartID column of the first row, a value of 2 for the second row, and so on. After you specify the column names, the statement uses a `VALUES` clause to specify the values for each column. The values for each row are enclosed in parentheses and separated by commas. Commas also separate the sets of values. In addition, you use `NULL` wherever a value is not known, which in this case is the PartMN column for each row.

You can view the values that you inserted in the Participants table by executing the following `SELECT` statement:

```
SELECT * FROM Participants;
```

The last tables to add data to are the four configured with foreign keys. Because the data in these tables references data in other tables, you had to populate those other tables first so that they contained the referenced data. Once the referenced tables contain the proper data, you can insert rows in the columns configured with the foreign keys. Because some of these tables have dependencies on each other, they too have to be populated in a specific order.

Try It Out — Inserting Data in the Foreign Key Tables

To insert data in these tables, take the following steps:

1. If it's not already open, open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

You should receive a message indicating that you switched to the DVDRentals database.

2. To insert a record in the DVDs table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDs
VALUES (NULL, 'White Christmas', DEFAULT, 2000, 'mt16', 's105', 'NR', 'f1', 's1');
```

You should receive a response saying that your statement executed successfully, affecting one row.

3. To insert the next record in the DVDs table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDs
   (DVDName, NumDisks, YearRlsd, MTypeID, StudID, RatingID, FormID, StatID)
VALUES ('What\'s Up, Doc?', 1, 2001, 'mt12', 's103', 'G', 'f1', 's2');
```

You should receive a response saying that your statement executed successfully, affecting one row.

4. To insert additional records in the DVDs table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDs
VALUES (NULL, 'Out of Africa', 1, 2000, 'mt11', 's101', 'PG', 'f1', 's1'),
(NULL, 'The Maltese Falcon', 1, 2000, 'mt11', 's103', 'NR', 'f1', 's2'),
(NULL, 'Amadeus', 1, 1997, 'mt11', 's103', 'PG', 'f1', 's2');
```

You should receive a response saying that your statement executed successfully, affecting three rows.

5. To insert the remaining records in the DVDs table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDs
   (DVDName, NumDisks, YearRlsd, MTypeID, StudID, RatingID, FormID, StatID)
VALUES
('The Rocky Horror Picture Show', 2, 2000, 'mt12', 's106', 'NR', 'f1', 's2'),
('A Room with a View', 1, 2000, 'mt11', 's107', 'NR', 'f1', 's1'),
('Mash', 2, 2001, 'mt12', 's106', 'R', 'f1', 's2');
```

You should receive a response saying that your statement executed successfully, affecting three rows.

6. To insert records in the DVDParticipant table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDParticipant
VALUES (3, 1, 'r102'),
(3, 4, 'r108'),
(3, 1, 'r103'),
(3, 2, 'r101'),
(3, 3, 'r101'),
(4, 6, 'r101'),
(1, 8, 'r101'),
(1, 9, 'r108'),
(1, 10, 'r102'),
(1, 11, 'r101'),
(1, 7, 'r101'),
(2, 5, 'r107');
```

You should receive a response saying that your statement executed successfully, affecting 12 rows.

7. To insert records in the Orders table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Orders (CustID, EmpID)
VALUES (1, 3),
(1, 2),
(2, 5),
(3, 6),
(4, 1),
(3, 3),
(5, 2),
(6, 4),
(4, 5),
(6, 2),
(3, 1),
(1, 6),
(5, 4);
```

You should receive a response saying that your statement executed successfully, affecting 13 rows.

8. To insert records in the Transactions table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Transactions (OrderID, DVDID, DateOut, DateDue)
VALUES (1, 1, CURDATE(), CURDATE()+3),
(1, 4, CURDATE(), CURDATE()+3),
(1, 8, CURDATE(), CURDATE()+3),
(2, 3, CURDATE(), CURDATE()+3),
(3, 4, CURDATE(), CURDATE()+3),
(3, 1, CURDATE(), CURDATE()+3),
(3, 7, CURDATE(), CURDATE()+3),
(4, 4, CURDATE(), CURDATE()+3),
(5, 3, CURDATE(), CURDATE()+3),
(6, 2, CURDATE(), CURDATE()+3),
(6, 1, CURDATE(), CURDATE()+3),
(7, 4, CURDATE(), CURDATE()+3),
(8, 2, CURDATE(), CURDATE()+3),
(8, 1, CURDATE(), CURDATE()+3),
(8, 3, CURDATE(), CURDATE()+3),
(9, 7, CURDATE(), CURDATE()+3),
(9, 1, CURDATE(), CURDATE()+3),
(10, 5, CURDATE(), CURDATE()+3),
(11, 6, CURDATE(), CURDATE()+3),
(11, 2, CURDATE(), CURDATE()+3),
(11, 8, CURDATE(), CURDATE()+3),
(12, 5, CURDATE(), CURDATE()+3);
(13, 7, CURDATE(), CURDATE()+3);
```

You should receive a response saying that your statement executed successfully, affecting 23 rows.

How It Works

In this exercise, you added data to the final four tables in the DVDRentals database. You populated these tables last because each one includes references (through foreign keys) to other tables in the database. In addition, the remaining tables include references to each other, so you had to add data to them in a specific order. You began the process by using the following INSERT statement:

```
INSERT INTO DVDs
VALUES (NULL, 'White Christmas', DEFAULT, 2000, 'mt16', 's105', 'NR', 'f1', 's1');
```

In this statement, you added one row to the DVDs table. Because you did not specify any columns, you included a value for each column in the table. To ensure that you inserted the correct data in the correct columns, you listed the values in the same order as they are listed in the table definition. For the first column, DVDID, you specified NULL. Because the column is configured with the AUTO_INCREMENT option, the value for this column was determined automatically. For the NumDisks column, you specified DEFAULT. As a result, the value 1 was inserted in this column because that is the default value defined on the column. For all other columns, you specified the values to be inserted in those columns.

The next INSERT statement that you used also inserted one row of data in the DVDs table; however, this statement specified the column names. As a result, the VALUES clause includes values only for the specified columns, as shown in the following statement:

```
INSERT INTO DVDs
   (DVDName, YearRlsd, MTypeID, StudID, RatingID, FormID, StatID)
VALUES ('What\'s Up, Doc?', 2001, 'mt12', 's103', 'G', 'f1', 's2');
```

By using this approach, you do not have to specify a NULL for the DVDID column or DEFAULT for the NumDiscs column. One other thing to note about this statement is that you used a backslash in the DVDName value to show that the apostrophe should be interpreted as a literal value. Without the backslash, the statement would not execute properly because the apostrophe would confuse the database engine.

The next statement you executed inserted several rows in the DVDs table, as the following statement demonstrates:

```
INSERT INTO DVDs
VALUES (NULL, 'Out of Africa', 1, 2000, 'mt11', 's101', 'PG', 'f1', 's1'),
(NULL, 'Maltese Falcon, The', 1, 2000, 'mt11', 's103', 'NR', 'f1', 's2'),
(NULL, 'Amadeus', 1, 1997, 'mt11', 's103', 'PG', 'f1', 's2');
```

Once again, because you didn't specify the column names, you had to provide NULL for the DVDID column. In addition, you provided a literal value (1) for the NumDisks column, even though that column is configured with a default of 1. Because you had to include a value, the value had to be DEFAULT or it had to be the literal figure.

You could have also created a statement that defined the columns to be inserted. In that case, you could have avoided repeating NULL for each row. Unless the value for the NumDisks column is always the default value, you would have had to include that column either way. The choice of whether to include column names or instead include all values depends on the table and how many rows you need to insert. For tables in which there are many rows of data and an AUTO_INCREMENT primary key, you're

usually better off specifying the column names and taking that approach, as you did when you inserted data in the Transactions table and the Orders table.

For example, when you inserted rows in the Transactions table, you specified the column names for every column except TransID, which is the primary key and which is configured with the AUTO_INCREMENT option. The following code shows just the first few rows of data of the INSERT statement that you used for the Transactions table:

```
INSERT INTO Transactions (OrderID, DVDID, DateOut, DateDue)
VALUES (1, 1, CURDATE(), CURDATE()+3),
(1, 4, CURDATE(), CURDATE()+3),
(1, 8, CURDATE(), CURDATE()+3),
(2, 3, CURDATE(), CURDATE()+3),
```

In this case, you can see the advantage of specifying the column names because you did not have to repeat NULL for each row. There is another aspect of this statement, though, that you haven't seen before. The DateOut value is determined by using the CURDATE() function. The function automatically returns the date on which the row is inserted in the table. The function is also used for the DateDue column. A value of 3 is added to the value returned by the function. As a result, the value inserted in the DateDue column is three days after the current date. Functions are very useful when creating expressions and defining values. For this reason, Chapter 9 focuses exclusively on the various functions supported by MySQL.

Using the <*set option*> Alternative of the INSERT Statement

The *<set option>* method for creating an INSERT statement provides you with an alternative to the *<values option>* method when you're adding only one row at a time to a table. The following syntax demonstrates how to use the *<set option>* alternative to create an INSERT statement:

```
<set option>::=
<table name>
SET <column name>={<expression> | DEFAULT}
   [{, <column name>={<expression> | DEFAULT}}...]
```

As the syntax shows, your INSERT statement must include the name of the table and a SET clause that specifies values for specific columns (all, of course, in addition to the required INSERT clause at the beginning of the statement). Although column names are not specified after the table name, as is the case for some statements that use the *<values option>* alternative, the column names are specified in the SET clause. For each value, you must specify the column name, an equal (=) sign, and an expression or the DEFAULT keyword. The *<expression>* option and the DEFAULT keyword work in the same way as they do for the previous INSERT statements that you saw. Also, if you include more than one column/value pair, you must separate them with a comma.

The *<set option>* alternative is most useful if you're providing values for only some of the columns and allowing default values to be used for the remaining columns. For example, suppose that you want to insert an additional row in the CDs table (the table used in the previous examples). If you used the *<values option>* method to create an INSERT statement, it would look similar to the following:

```
INSERT INTO CDs (CDName, Copyright, NumberDisks,
   NumberInStock, NumberOnReserve, NumberAvailable, CDType)
VALUES ('Blues on the Bayou', 1998, DEFAULT,
   4, 1, NumberInStock-NumberOnReserve, 'Blues');
```

Notice that each column is specified after the table name and that their respective values are included in the `VALUES` clause. You could rewrite the statement by using the *<set option>* alternative, as shown in the following example:

```
INSERT DELAYED INTO CDs
SET CDName='Blues on the Bayou', Copyright=1998,
   NumberDisks=DEFAULT, NumberInStock=4, NumberOnReserve=1,
   NumberAvailable=NumberInStock-NumberOnReserve, CDType='Blues';
```

As you can see, the column names, along with their values, are specified in the `SET` clause. You do not have to enclose them in parentheses, although you do need to separate them with commas. The main advantage to using this method is that it simplifies matching values to column names, without having to refer back and forth between clauses. As a result, you ensure that you always match the proper value to the correct column. If you plan to provide values for each column, you save yourself keystrokes by using the *<values option>* alternative because you don't need to specify column names when values are provided for all columns. In addition, the *<values option>* alternative allows you to insert multiple rows with one statement. The *<set option>* alternative does not.

In the previous Try It Out sections, you used the *<values option>* alternative to insert data in the DVDRentals database. In this exercise, you use the *<set option>* alternative to insert a row in the DVDs table.

Try It Out — Using the <*set option*> Alternative to Insert Data in the DVDRentals Database

Follow these steps to insert the data:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To insert the record in the DVDs table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDs
SET DVDName='Some Like It Hot', YearRlsd=2001, MTypeID='mt12',
   StudID='s108', RatingID='NR', FormID='f1', StatID='s2';
```

   You should receive an error message stating that a foreign key constraint failed because the StudID value does not exist in the referenced parent table (Studios).

3. To insert the necessary record in the Studios table, type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO Studios
VALUES ('s108', 'Metro-Goldwyn-Mayer');
```

   You should receive a response saying that your statement executed successfully, affecting one row.

4. Now you should be able to insert the record in the DVDs table. Type the following `INSERT` statement at the mysql command prompt, and then press Enter:

```
INSERT INTO DVDs
SET DVDName='Some Like It Hot', YearRlsd=2001, MTypeID='mt12',
   StudID='s108', RatingID='NR', FormID='f1', StatID='s2';
```

You should receive a response saying that your statement executed successfully and that one row was affected.

How It Works

The first `INSERT` statement that you used in this exercise attempted to insert a row in the DVDs table in the DVDRentals database, as shown in the following statement:

```
INSERT INTO DVDs
SET DVDName='Some Like It Hot', YearRlsd=2001, MTypeID='mt12',
   StudID='s108', RatingID='NR', FormID='f1', StatID='s2';
```

As you would expect with the *<set option>* alternative, the column names and values are specified in the `SET` clause. This `INSERT` statement failed, though, because you attempted to insert a row that contained a StudID value of s108. As you recall, you defined a foreign key on the StudID column in the DVDs table. The column references the StudID column in the Studios table. Because the Studios table doesn't contain the referenced value of s108 in the StudID column, you could not insert a row in DVDs that includes a StudID value of s108.

To remedy this situation, the following INSERT statement inserts the necessary value in the Studios table:

```
INSERT INTO Studios
VALUES ('s108', 'Metro-Goldwyn-Mayer');
```

Because the Studios table contains only two columns and because you needed to provide values for both those columns, the *<values option>* alternative adds the row to the Studios table. Once you completed this task, you then used the original `INSERT` statement to add the same row to the DVDs table. This time the statement succeeded.

Using a REPLACE Statement to Add Data

In addition to using an `INSERT` statement to add data to a table, you can also use a `REPLACE` statement. A `REPLACE` statement is similar to an `INSERT` statement in most respects. The main difference between the two is in how values in a primary key column or a unique index are treated. In an `INSERT` statement, if you try to insert a row that contains a unique index or primary key value that already exists in the table, you aren't able to add that row. A `REPLACE` statement, however, deletes the old row and adds the new row.

Another method that programmers have used to support the same functionality as that of the `REPLACE` *statement is to test first for the existence of a row and then use an* `UPDATE` *statement if the row exists or an* `INSERT` *statement if the row doesn't exist. The way in which you would implement this method depends on the programming language that you're using. For information on how to set up the conditions necessary to execute the* `UPDATE` *and* `INSERT` *statements in this way, see the documentation for that particular language.*

Despite the issue of primary key and unique index values, the syntax for the REPLACE statement basically contains the same elements as the INSERT statement, as the following syntax shows:

```
<replace statement>::=
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
{<values option> | <set option> | <select option>

<values option>::=
<table name> [(<column name> [{, <column name>}...])]
VALUES ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])
   [{, ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])}...]

<set option>::=
<table name>
SET <column name>={<expression> | DEFAULT}
   [{, <column name>={<expression> | DEFAULT}}...]

<select option>::=
<table name> [(<column name> [{, <column name>}...])]
<select statement>
```

As you can see, the main difference between the INSERT syntax and the REPLACE syntax is that you use the REPLACE keyword rather than the INSERT keyword. In addition, the REPLACE statement doesn't support the IGNORE option. The REPLACE statement does include the same three methods for creating the statement: the *<values option>* alternative, the *<set option>* alternative, and the *<select option>* alternative. As with the INSERT statement, the discussion here includes only the first two alternatives. Chapter 11 discusses the *<select option>* alternative.

> *Although the* REPLACE *statement and the* INSERT *statement are nearly identical, the difference between the two can be a critical one. By using the* REPLACE *statement, you risk overwriting important data. Use caution whenever executing a* REPLACE *statement.*

Using the <*values option*> Alternative of the REPLACE Statement

As the following syntax shows, the *<values option>* alternative for the REPLACE statement is the same as that alternative for the INSERT statement:

```
<values option>::=
<table name> [(<column name> [{, <column name>}...])]
VALUES ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])
   [{, ({<expression> | DEFAULT} [{, {<expression> | DEFAULT}}...])}...]
```

As you can see, this version of the INSERT statement contains all the same elements, so have a look at a couple of examples to help demonstrate how this statement works. The examples are based on the following table definition:

```
CREATE TABLE Inventory
(
   ProductID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   NumberInStock SMALLINT UNSIGNED NOT NULL,
   NumberOnOrder SMALLINT UNSIGNED NOT NULL,
   DateUpdated DATE
);
```

The main characteristic to note about the Inventory table is that the ProductID column is configured as the primary key. It doesn't use the AUTO_INCREMENT option, so you have to provide a value for this column. The following REPLACE statement adds values to each column in the Inventory table:

```
REPLACE LOW_PRIORITY INTO Inventory
VALUES (101, 20, 25, '2004-10-14');
```

You can view the values that you inserted in the Inventory table by executing the following SELECT statement:

```
SELECT * FROM Inventory;
```

Note that the LOW_PRIORITY option, the INTO keyword, the table name, and the values in the VALUES clause are specified exactly as they would be in an INSERT statement. If you were to execute this statement and no rows in the table contain a ProductID value of 101, the statement would be inserted in the table and you would lose no data. Suppose that you then executed the following REPLACE statement:

```
REPLACE LOW_PRIORITY INTO Inventory
VALUES (101, 10, 25, '2004-10-16');
```

This statement also includes a ProductID value of 101. As a result, the original row with that ProductID value would be deleted and the new row would be inserted. Although this might be the behavior that you expected, what if you had meant to insert a different ProductID value? If you had, the original data would be lost and the new data would be inaccurate. For this reason, you should be very cautious when using the REPLACE statement. For the most part, you are better off using an INSERT statement.

In the following exercise, you try out a REPLACE statement by inserting a row in the Studios table.

Try It Out — Using the REPLACE...VALUES Form to Insert Data in the DVDRentals Database

To complete this exercise, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To insert the record in the Studios table, type the following REPLACE statement at the mysql command prompt, and then press Enter:

```
REPLACE Studios (StudID, StudDescrip)
VALUES ('s109', 'New Line Cinema, Inc.');
```

   You should receive a response saying that your statement executed successfully, affecting one row.

3. Now insert another record in the Studios table. This record includes the same primary key value as the last row you inserted, but the studio name is slightly different. Type the following REPLACE statement at the mysql command prompt, and then press Enter:

```
REPLACE Studios
VALUES ('s109', 'New Line Cinema');
```

You should receive a response saying that your statement executed successfully, affecting two rows.

How It Works

The first REPLACE statement inserted a new row in the Studios table, as shown in the following statement:

```
REPLACE Studios (StudID, StudDescrip)
VALUES ('s109', 'New Line Cinema, Inc.');
```

The new row contained a StudID value of s109 and a StudDescrip value of New Line Cinema, Inc. After executing this statement, you then executed the following REPLACE statement:

```
REPLACE Studios
VALUES ('s109', 'New Line Cinema');
```

For this statement, you did not specify the column names because you inserted a value for each column. You again inserted a row that contained a StudID value of s109. Because the StudID column is configured as the primary key, that last statement replaced the row that the first statement created. As a result, the StudDescrip column now has a value of New Line Cinema.

You can view the values that you inserted in the Studios table by executing the following SELECT statement:

```
SELECT * FROM Studios;
```

Using the <*set option*> Alternative of the REPLACE Statement

As with the *<values option>* alternative, the *<set option>* alternative of the REPLACE statement also includes the same elements that you use in the INSERT statement, as shown in the following syntax:

```
<set option>::=
<table name>
SET <column name>={<expression> | DEFAULT}
   [{, <column name>={<expression> | DEFAULT}}...]
```

The SET clause in the REPLACE statement includes the column names and the values for those columns. Take a look at a couple of examples that help demonstrate this. The following REPLACE statement uses the *<values option>* alternative to add a row to the Inventory table (which is the same table used in the previous REPLACE examples):

```
REPLACE INTO Inventory (ProductID, NumberInStock, DateUpdated)
VALUES (107, 16, '2004-11-30');
```

Notice that values are specified for the ProductID, NumberInStock, and DateUpdated columns, but not for the NumberOnOrder column. Because this column is configured with an integer data type and because the column does not permit null values, a 0 value is added to the column. (When a value is not provided for a column configured with an integer data type, a 0 is inserted if that column does not permit null values and no default is defined for the column. For more information about integer data types, see Chapter 5.)

You could insert the same data shown in the preceding example by using the following REPLACE statement:

```
REPLACE INTO Inventory
SET ProductID=107, NumberInStock=16, DateUpdated='2004-11-30';
```

Notice that a SET clause is used this time, but the column names and values are the same. The SET clause provides the same advantages and disadvantages as using the SET clause in an INSERT statement. The main consideration is, of course, that you do not accidentally overwrite data that you intended to retain.

In the following exercise, you try out the *<set option>* alternative of the REPLACE statement. You use this form of the statement to add two rows to the MovieTypes table.

Try It Out — Using the <*set option*> Alternative to Insert Data in the DVDRentals Database

To add these rows, follow these steps:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To insert the record in the MovieTypes table, type the following REPLACE statement at the mysql command prompt, and then press Enter:

```
REPLACE MovieTypes
SET MTypeID='mt17', MTypeDescrip='Foreign-subtitled';
```

   You should receive a response saying that your statement executed successfully, affecting one row.

3. Now insert another record in the MovieTypes table. This record includes the same primary key value as the last row you inserted, but the name is slightly different. Type the following REPLACE statement at the mysql command prompt, and then press Enter:

```
REPLACE MovieTypes
SET MTypeID='mt17', MTypeDescrip='Foreign';
```

   You should receive a response saying that your statement executed successfully, affecting two rows. MySQL reports that two rows are affected because it treats the original row as a deletion and the updated row as an insertion.

How It Works

The first REPLACE statement adds one row to the MovieTypes table, as shown in the following statement:

```
REPLACE MovieTypes
SET MTypeID='mt17', MTypeDescrip='Foreign-subtitled';
```

The added row includes an MTypeID value of mt17 and an MTypeDescrip value of Foreign-subtitled. Because values are defined for both columns in this table, you could just as easily use the *<values option>* alternative of the REPLACE statement. That way, you would not have to specify the column names.

After adding the row to the MovieTypes table, you executed the following REPLACE statement:

```
REPLACE MovieTypes
SET MTypeID='mt17', MTypeDescrip='Foreign';
```

Notice that you've specified the same MTypeID value (mt17), but a slightly different MTypeDescrip value. The question then becomes whether your intent is to replace the original row or to include two rows in the table, one for Foreign films with subtitles and one for other foreign films. Again, this points to the pitfalls of using a REPLACE statement rather than an INSERT statement.

Updating Data in a MySQL Database

Now that you have a foundation in how to insert data in a MySQL database, you're ready to learn how to update that data. The primary statement used to modify data in a MySQL database is the UPDATE statement. The following syntax shows the elements included in an UPDATE statement:

```
<update statement>::=
UPDATE [LOW_PRIORITY] [IGNORE]
<single table update> | <joined table update>

<single table update>::=
<table name>
SET <column name>=<expression> [{, <column name>=<expression>}...]
[WHERE <where definition>]
[ORDER BY <column name> [ASC | DESC] [{, <column name > [ASC | DESC]}...]]
[LIMIT <row count>]

<joined table update>::=
<table name> [{, <table name>}...]
SET <column name>=<expression> [{, <column name>=<expression>}...]
[WHERE <where definition>]
```

The first line of the syntax for the UPDATE statement contains the mandatory UPDATE keyword along with the LOW_PRIORITY option and the IGNORE option, both of which you've seen in the INSERT statement. You should use the LOW_PRIORITY option when you want to delay the execution of the UPDATE statement until no other client connections are accessing that targeted table. You should use the IGNORE option if you want an update to continue even if duplicate primary key and unique index values are found. (The row with the duplicate value is not updated.)

In addition to the UPDATE clause, the syntax for the UPDATE statement specifies two statement alternatives: the *<single table update>* alternative and the *<joined table update>* alternative. A joined table refers to a table that is joined to another table in an SQL statement, such as the UPDATE statement. The join is based on the foreign key relationship established between these two tables. You can use this relationship to access related data in both tables in order to perform an operation on the data in the related tables. Although Chapter 10 discusses joins in greater detail, this chapter includes a brief discussion of joins because they are integral to the *<joined table update>* method of updating a table. Before getting into a discussion about updating joined tables, this chapter first discusses updating a single table.

Using an UPDATE Statement to Update a Single Table

To update a single table in a MySQL database, in which no join conditions are taken into account in order to perform that update, you should create an `UPDATE` statement that uses the *`<single table update>`* alternative, which is shown in the following syntax:

```
<single table update>::=
<table name>
SET <column name>=<expression> [{, <column name>=<expression>}...]
[WHERE <where definition>]
[ORDER BY <column name> [ASC | DESC] [{, <column name> [ASC | DESC]}...]]
[LIMIT <row count>]
```

As the syntax shows, you must specify a table name and a `SET` clause. The `SET` clause includes, at the very least, a column name and an associated expression, connected by an equal (=) sign. The information sets a value for a particular column. If you want to include more than one column, you must separate the column/expression pairs with commas.

In addition to the of the *`<single table update>`* alternative's required elements, you can choose to include several additional clauses in your statement. The first of these — the `WHERE` clause — determines which rows in a table are updated. Without a `WHERE` clause, all tables are updated, which might sometimes be what you want, but most likely you want to use a `WHERE` clause to qualify your update. The `WHERE` clause includes one or more conditions that define the extent of the update.

> *Because the* `WHERE` *clause is such an integral part of a* `SELECT` *statement, it is discussed in detail in Chapter 7; however, this chapter includes information about the clause so that you have a better understanding of the* `UPDATE` *statement. But know that, after you gain a more in-depth knowledge of the* `WHERE` *clause in Chapter 7, you'll be able to refine your* `UPDATE` *statements to an even greater degree.*

The next optional clause in the `UPDATE` statement is the `ORDER BY` clause. The `ORDER BY` clause allows you to specify that the rows updated by your `UPDATE` statement are updated in a specified order according to the values in the column or columns you specify in the clause. If you specify more than one column, you must separate them by a comma, in which case, the rows are updated based on the first column specified, then the second, and so on. In addition, for each column that you include in the clause, you can specify the `ASC` option or the `DESC` option. The `ASC` option means that the rows should be updated in ascending order, based on the column values. The `DESC` option means that the rows should be updated in descending order. If you specify neither option, the rows are updated in ascending order.

In addition to the `ORDER BY` clause, you can include a `LIMIT` clause that limits the number of rows updated to the value specified in the `LIMIT` clause. For example, if your `UPDATE` statement would normally update 10 rows in a table, but you specify a `LIMIT` clause with a value of 5, only the first five rows are updated. Because of a LIMIT clause's nature, it is best suited to use in conjunction with an `ORDER BY` clause.

To understand how these clauses work, take a look at a few examples. The examples in this section and the next section, which discusses updating joined tables, are based on two tables. The first table is the Books table, which is shown in the following table definition:

```
CREATE TABLE Books
(
   BookID SMALLINT NOT NULL PRIMARY KEY,
   BookName VARCHAR(40) NOT NULL,
   InStock SMALLINT NOT NULL
)
ENGINE=INNODB;
```

The following `INSERT` statement populates the Books table:

```
INSERT INTO Books
VALUES (101, 'Noncomformity: Writing on Writing', 12),
(102, 'The Shipping News', 17),
(103, 'Hell\'s Angels', 23),
(104, 'Letters to a Young Poet', 32),
(105, 'A Confederacy of Dunces', 6),
(106, 'One Hundred Years of Solitude', 28);
```

The next table is the Orders table, which includes a foreign key that references the Books table, as shown in the following table definition:

```
CREATE TABLE Orders
(
   OrderID SMALLINT NOT NULL PRIMARY KEY,
   BookID SMALLINT NOT NULL,
   Quantity TINYINT (40) NOT NULL DEFAULT 1,
   DateOrdered TIMESTAMP,
   FOREIGN KEY (BookID) REFERENCES Books (BookID)
)
ENGINE=INNODB;
```

The following INSERT statement populates the Orders table:

```
INSERT INTO Orders
VALUES (1001, 103, 1, '2004-10-12 12:30:00'),
(1002, 101, 1, '2004-10-12 12:31:00'),
(1003, 103, 2, '2004-10-12 12:34:00'),
(1004, 104, 3, '2004-10-12 12:36:00'),
(1005, 102, 1, '2004-10-12 12:41:00'),
(1006, 103, 2, '2004-10-12 12:59:00'),
(1007, 101, 1, '2004-10-12 13:01:00'),
(1008, 103, 1, '2004-10-12 13:02:00'),
(1009, 102, 4, '2004-10-12 13:22:00'),
(1010, 101, 2, '2004-10-12 13:30:00'),
(1011, 103, 1, '2004-10-12 13:32:00'),
(1012, 105, 1, '2004-10-12 13:40:00'),
(1013, 106, 2, '2004-10-12 13:44:00'),
(1014, 103, 1, '2004-10-12 14:01:00'),
(1015, 106, 1, '2004-10-12 14:05:00'),
(1016, 104, 2, '2004-10-12 14:28:00'),
(1017, 105, 1, '2004-10-12 14:31:00'),
(1018, 102, 1, '2004-10-12 14:32:00'),
(1019, 106, 3, '2004-10-12 14:49:00'),
(1020, 103, 1, '2004-10-12 14:51:00');
```

Notice that the values added to the BookID column in the Orders table include only values that exist in the BookID column in the Books table. The BookID column in Orders is the referencing column, and the BookID column in Books is the referenced column.

After creating the tables and adding data to those tables, you can modify that data. The following UPDATE statement modifies values in the InStock column of the Books table:

```
UPDATE Books
SET InStock=InStock+10;
```

This statement includes only the required elements of an UPDATE statement. This includes the UPDATE keyword, the name of the table, and a SET clause that specifies one column/expression pair. The expression in this case is made up of the InStock column name, a plus (+) arithmetic operator, and a literal value of 10. As a result, the existing value in the InStock column increases by 10. Because you specify no other conditions in the UPDATE statement, all rows in the table update. This approach might be fine in some cases, but in all likelihood, most of your updates should be more specific than this. As a result, you want to qualify your statements so that only certain rows in the table update, rather than all rows.

The method used to qualify an UPDATE statement is to add a WHERE clause to the statement. The WHERE clause provides the specifics necessary to limit the update. For example, the following UPDATE statement includes a WHERE clause that specifies that only rows with an OrderID value of 1001 should be updated:

```
UPDATE Orders
SET Quantity=2
WHERE OrderID=1001;
```

As you can see, the WHERE clause allows you to be much more specific with your updates. In this case, you specify the OrderID column, followed by an equal (=) sign, and then followed by a value of 1001. As a result, the value in the OrderID column must be 1001 in order for a row to be updated. Because this is the primary key column, only one row contains this value, so that is the only row updated. In addition, the SET clause specifies that the value in the Quantity column be set to 2. The result is that the Quantity column in the row that contains an OrderID value of 1001 is updated, but no other rows are updated.

As you have seen, another method that you can use to update a value in a column is to base the update on the current value in that column. For example, in the following UPDATE statement, the SET clause uses the Quantity column to specify the new value for that column:

```
UPDATE Orders
SET Quantity=Quantity+1
WHERE OrderID=1001;
```

In this statement, the SET clause specifies that the new value for the Quantity column should equal the current value plus one. For example, if the current value is 2, it increases to 3. Also, because the UPDATE statement is qualified by the use of a WHERE clause, only the row with an OrderID value of 1001 increases the value in the Quantity column by a value of 1.

In the last two examples, the WHERE clause specifies that only one record should be updated (OrderID=1001). If the OrderID column were not the primary key and duplicate values were permitted in that column, it would be conceivable that more than one row would be updated, which is often the

case in an UPDATE statement. For example, the following statement updates all rows with an InStock value of less than 30:

```
UPDATE LOW_PRIORITY Books
SET InStock=InStock+10
WHERE InStock<30;
```

In this statement, the SET clause specifies that all values in the InStock column should be increased by 10. (The addition [+] operator is used to add 10 to the value in the InStock column.) The WHERE clause limits that update only to those columns that contain an InStock value of less than 30. (The less than [<] operator indicates that the value in the InStock column must be less that 30.) As a result, the number of rows updated equals the number of rows with an InStock value less than 30. (Chapter 8 describes the operators available in MySQL and how to use those operators in your SQL statements.)

An UPDATE statement can be further qualified by the use of the ORDER BY clause and the LIMIT clause. For example, suppose you want to update the Orders table so that orders that contain a BookID value of 103 are increased by 1. You want to limit the increase, though, to the last five orders for that book. You could use the following UPDATE statement to modify the table:

```
UPDATE Orders
SET Quantity=Quantity+1
WHERE BookID=103
ORDER BY DateOrdered DESC
LIMIT 5;
```

By now, you should be familiar with the first three lines of code. In the first line, the UPDATE clause specifies the Orders table. In the second line, the SET clause specifies that the values in the Quantity column should be increased by 1. In the third line, the WHERE clause specifies that only rows that contain a BookID value of 103 should be updated.

The ORDER BY clause then further qualifies the statement by ordering the rows to be updated by the values in the DateOrdered column. Notice the use of the DESC option, which means that the values are sorted in descending order. Because this is a TIMESTAMP column, descending order means that the most recent orders are updated first. Because the LIMIT clause specifies a value of 5, only the first five rows are updated.

As you can see, the ORDER BY and LIMIT clauses are useful only in very specific circumstances, but the WHERE and SET clauses provide a considerable amount of flexibility in defining UPDATE statements.

Now that you have seen examples of how an UPDATE statement works, it's time to try a couple out for yourself. In this exercise you create UPDATE statements that modify values in the DVDs table of the DVDRentals database.

Try It Out — Using UPDATE Statements to Modify Data in a Single Table

Follow these steps to perform updates to the DVDs table:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

You should receive a message indicating that you switched to the DVDRentals database.

2. To update the record in the DVDs table, type the following `UPDATE` statement at the mysql command prompt, and then press Enter:

```
UPDATE DVDs
SET StudID='s110'
WHERE DVDID=9;
```

You should receive an error message stating that a foreign key constraint has failed. This happens because the StudID value does not exist in the referenced parent table (Studios).

3. Now try updating the table in a different way. To update the record in the DVDs table, type the following `UPDATE` statement at the mysql command prompt, and then press Enter:

```
UPDATE DVDs
SET StatID='s1'
WHERE DVDID=9;
```

This time, you should receive a response saying that your statement executed successfully, affecting one row.

4. To update the same record in the DVDs table, type the following `UPDATE` statement at the mysql command prompt, and then press Enter:

```
UPDATE DVDs
SET StatID='s3', MTypeID='mt13'
WHERE DVDID=9;
```

You should receive a response saying that your statement executed successfully, affecting one row.

How It Works

In this exercise, you created three `UPDATE` statements that attempted to update the DVDs table. The first statement tried to update the StudID value in the row that contained a DVDID of 9, as shown in the following statement:

```
UPDATE DVDs
SET StudID='s110'
WHERE DVDID=9;
```

Your attempt to execute this statement failed because you tried to update the StudID value to a value that didn't exist in the referenced column in the parent table (Studios). Had the Studios table contained a row with a StudID value of s110, this statement would have succeeded.

In this next `UPDATE` statement that you created, you tried to change the StatID value of the row that contains a DVDID value of 9, as shown in the following statement:

```
UPDATE DVDs
SET StatID='s1'
WHERE DVDID=9;
```

The `SET` clause in this statement specifies that the StatID column should be changed to a value of s1. When you executed the statement, the value changed successfully because the value existed in the referenced column of the parent table (Status). In addition, because the `WHERE` clause specifies that only rows

with a DVDID value of 9 should be updated, only one row changed because the DVDID column is the primary key, so only one row contained that value.

The final UPDATE statement that you created in this exercise updated two values in the DVDs table, as the following statement shows:

```
UPDATE DVDs
SET StatID='s3', MTypeID='mt13'
WHERE DVDID=9;
```

In this statement, you specified that the StatID value should be changed to s3 and that the MTypeID value should be changed to mt13, both acceptable values in the referenced column. Notice that a comma separates the expressions in the SET clause. In addition, as with the previous statement, the updates applied only to the row that contained a DVDID value of 9.

Using an UPDATE Statement to Update Joined Tables

In the example UPDATE statements that you've looked at so far, you updated individual tables without joining them to other tables. Although the tables contained foreign keys that referenced other tables, you specified no join conditions in the UPDATE statements. For a join condition to exist, it must be explicitly defined in the statement.

As you saw earlier in the chapter, the UPDATE statement includes an option that allows you to update joined tables. The following syntax shows the basic elements that are used to create an UPDATE statement that modifies joined tables:

```
<joined table update>::=
<table name> [{, <table name>}...]
SET <column name>=<expression> [{, <column name>=<expression>}...]
[WHERE <where definition>]
```

As you can see, this syntax contains many of the same elements as the syntax used to update a single table that is not specified in a join. One difference, however, is that you can specify more than one table in a statement that uses the *<joined table update>* alternative. In addition, this method of updating a table does not allow you to specify an ORDER BY clause or a LIMIT clause.

> *Generally, using an* UPDATE *statement to modify data in multiple tables is not recommended for* InnoDB *tables. Instead, you should rely on the* ON DELETE *and* ON CASCADE *options specified in the foreign key constraints of the table definitions.*

When updating a joined table, you must specify the tables in the join, qualify the column names with table names, and define the join condition in the WHERE clause, as shown in the following UPDATE statement:

```
UPDATE Books, Orders
SET Books.InStock=Books.InStock-Orders.Quantity
WHERE Books.BookID=Orders.BookID
   AND Orders.OrderID=1002;
```

Take a look at this statement one clause at a time. The UPDATE clause includes the name of both the Books table and the Orders table. Although you are updating only the Books table, you must specify both tables because both of them are included in the joined tables. Notice that a comma separates the table names.

The SET clause in this statement uses qualified column names to assign an expression to the InStock column. A *qualified column name* is one that is preceded by the name of the table and a period. This allows MySQL (and you) to distinguish between columns in different tables that have the same name. For example, the first column listed (Books.InStock) refers to the InStock column in the Books table. Qualified names are required whenever column names can be confused.

According to the SET clause, the UPDATE statement should set the value of the Books.InStock column to equal its current value less the value in the Quantity column of the Orders table. For example, if a value in the Books.InStock column is 6 and the value in the Orders.Quantity column is 2, the new value of the Books.InStock column should be 4.

The next clause in the UPDATE statement is the WHERE clause. In this clause, you join the two tables by matching values in the BookID columns in each table: Books.BookID=Orders.BookID. By associating the tables through related columns in this way, you are creating a type of join. (A *join* is a condition defined in a SELECT, UPDATE, or DELETE statement that links together two or more tables. You learn more about joins in Chapter 10.)

In the preceding example, the joined columns indicate that values in the Books.BookID column should match values in the Orders.BookID column. As a result, the UPDATE statement affects only rows with matching values. This is how the join condition is defined.

The WHERE clause further qualifies how rows are updated by specifying that the value in the OrderID column of the Orders table must equal 1002. Because the AND keyword connects these two conditions (the Books.BookID=Orders.BookID condition and the Orders.OrderID=1002 condition), a row is updated only if both conditions are met. In other words, the Books.BookID value must equal the Orders.OrderID value *and* the Orders.OrderID value must equal 1002. The Books.InStock value is updated when both these conditions are met.

When you update values in a joined table, you can update more than one value at a time, as the following example demonstrates:

```
UPDATE Books, Orders
SET Orders.Quantity=Orders.Quantity+2,
   Books.InStock=Books.InStock-2
WHERE Books.BookID=Orders.BookID
   AND Orders.OrderID = 1002;
```

In this UPDATE statement, the same conditions hold true as in the previous statement: the Books.BookID value must equal the Orders.OrderID value, *and* the Orders.OrderID value must equal 1002. However, the SET clause is a little different in this statement. If a row meets the conditions specified in the WHERE clause, the Orders.Quantity value is increased by 2, and the Books.InStock value is decreased by 2. As you can see, you can update both tables specified in the join condition.

In the following exercise, you update values in a joined table in the DVDRentals database.

Try It Out Using UPDATE Statements to Modify Data in Joined Tables

The tables you modify include DVDs and Studios, as shown in the following steps.

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you have switched to the DVDRentals database.

2. To update the record in the DVDs table, type the following `UPDATE` statement at the mysql command prompt, and then press Enter:

```
UPDATE DVDs, Studios
SET DVDs.StatID='s2'
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='Metro-Goldwyn-Mayer';
```

   You should receive a response saying that your statement executed successfully and that one row was affected.

How It Works

In this exercise, you created the following `UPDATE` statement to modify values in StatID column of the DVDs table:

```
UPDATE DVDs, Studios
SET DVDs.StatID='s2'
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='Metro-Goldwyn-Mayer';
```

The statement begins with an `UPDATE` clause that specifies the names of the tables that participate in the join condition. Next, the `SET` clause specifies that the value in the StatID column of the DVDs table should be set to s2. The last clause in the statement is a `WHERE` clause that defines the join condition. In this case, the StudID column in the DVDs table is joined to the StudID column of the Studios table.

The join defined in the `WHERE` clause represents one of two conditions defined in the clause. The second condition — `Studios.StudDescrip='Metro-Goldwyn-Mayer'` — means that the StudDescrip column of the Studios table must be Metro-Goldwyn-Mayer. This means that, in order for a row to be updated, the value in the DVDs.StudID column must equal the value in the Studios.StudID column *and* the Studios.StudDescrip column must contain the value Metro-Goldwyn-Mayer. If both of these conditions are met, the DVDs.StatID column is set to s2. In practical terms, this statement specifies that the status for any DVD released by Metro-Goldwyn-Mayer should be set to s2, which means that the DVD is available to rent. You might run a statement like this if all your MGM movies have been put on hold for promotional reasons and are now available.

In this exercise, you used an `UPDATE` *statement to modify data in multiple* `InnoDB` *tables, even though this method is normally not recommended. The exercise had you perform these updates so that you could try out how these types of* `UPDATE` *statements work.*

Deleting Data from a MySQL Database

It is inevitable that, after entering data into a database, some of it will have to be deleted. In most cases, you can delete data only if a foreign key column is not referencing it. In that case, you must first delete or modify the referencing value, and then you can delete the row that contains the referenced value.

When you delete data from a table, you must delete one or more entire rows at a time. You cannot delete only a part of a row. MySQL supports two statements that you can use to delete data from a database: the `DELETE` statement and the `TRUNCATE` statement.

Using a DELETE Statement to Delete Data

The `DELETE` statement is the primary statement that you use to remove data from a table. The syntax for the statement is as follows:

```
<delete statement>::=
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
{<single table delete> | <from join delete> | <using join delete>

<single table delete>::=
FROM <table name>
[WHERE <where definition>]
[ORDER BY <column name> [ASC | DESC] [{, <column name> [ASC | DESC]}...]]
[LIMIT <row count>]

<from join delete>::=
<table name>[.*] [{, <table name>[.*]}...]
FROM <table name> [{, <table name>}...]
[WHERE <where definition>]

<using join delete>::=
FROM <table name>[.*] [{, <table name>[.*]}...]
USING <table name> [{, <table name>}...]
[WHERE <where definition>]
```

The first line of the statement includes the mandatory `DELETE` keyword, along with the `LOW_PRIORITY`, `QUICK`, and `IGNORE` options. You've seen the `LOW_PRIORITY` option used in `INSERT`, `REPLACE`, and `UPDATE` statements, and you've seen the `IGNORE` option used in `INSERT` and `UPDATE` statements. If you specify `LOW_PRIORITY`, the `DELETE` statement does not execute until no client connections are accessing the target table. If you specify `IGNORE`, errors are not returned when trying to delete rows; rather, warnings are provided if a row cannot be deleted.

The other option in the `DELETE` clause is `QUICK`. This option applies only to `MyISAM` table. When you use the `QUICK` option, the MyISAM storage engine does not merge certain components of the index during a delete operation, which may speed up some operations.

The next line of code in the `DELETE` statement syntax specifies three alternatives that you can use when creating a statement: the *`<single table delete>`* alternative, the *`<from join delete>`* alternative, and the *`<using join delete>`* alternative. The *`<from join delete>`* alternative refers to join conditions specified in the statement's `FROM` clause, and the *`<using join delete>`* alternative refers to join conditions that you can specify in the `USING` clause. The following sections discuss all three alternatives.

Deleting Data from a Single Table

The first type of delete that you can perform is based on the *<single table delete>* alternative. You can use this method to delete rows from a table not defined in a join condition, as shown in the following syntax:

```
<single table delete>::=
FROM <table name>
[WHERE <where definition>]
[ORDER BY <column name> [ASC | DESC] [{, <column name> [ASC | DESC]}...]]
[LIMIT <row count>]
```

As the syntax shows, the *<single table delete>* alternative requires a `FROM` clause that defines the name of the table. Optionally, you can also add a `WHERE`, an `ORDER BY`, or a `LIMIT` clause. These clauses work the same way they do in an `UPDATE` statement. The `WHERE` clause includes one or more conditions that define the extent of the delete operation. The `ORDER BY` clause sorts rows according to the column or columns specified in the clause. The `LIMIT` clause limits the number of rows to be deleted to the number specified in the clause.

To better illustrate how to use the `DELETE` statement, consider a few examples. The first example is a `DELETE` statement that contains only the required components of the statement, as shown in the following statement:

```
DELETE FROM Orders;
```

As you can see, the statement specifies only the `DELETE FROM` keywords and the table name. Because a `WHERE` clause does not qualify the statement, all rows are deleted from the table. Although this might be your intent, it might also result in an unintentional loss of data. As a result, most `DELETE` statements include a `WHERE` clause that defines which rows should be deleted. For example, the following statement deletes only rows that contain an OrderID value of 1020:

```
DELETE FROM Orders
WHERE OrderID=1020;
```

There `WHERE` clause makes the statement much more specific, and now only the applicable rows are deleted. Rows that do not contain an OrderID value of 1020 are left alone.

A `DELETE` statement can be further qualified by using an `ORDER BY` and a `LIMIT` clause, as shown in the following example:

```
DELETE LOW_PRIORITY FROM Orders
WHERE BookID=103
ORDER BY DateOrdered DESC
LIMIT 1;
```

In this statement, the rows to be deleted (those with a BookID of 103) are sorted according to the DateOrdered column, in descending order, meaning that the rows with the most recent dates are deleted first. The `LIMIT` clause restricts the deletion to only one row. As a result, only the most recent order for the book with the BookID value of 103 is deleted.

Indeed, deleting data from a table that is not joined to another table is a relatively straightforward process, and several examples demonstrate the ease with which you can remove information from a database. The following Try It Out shows you how to use the `DELETE` statement to remove data from the MovieTypes table in the DVDRentals database.

Try It Out — Using DELETE Statements to Delete Data from a Single Table in the DVDRentals Database

The following steps walk you though the delete operation:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To delete the record in the MovieTypes table, type the following `DELETE` statement at the mysql command prompt, and then press Enter:

```
DELETE FROM MovieTypes
WHERE MTypeID='mt18';
```

   You should receive a response saying that your statement executed successfully but that no rows were affected.

3. To delete the record in the MovieTypes table, type the following `DELETE` statement at the mysql command prompt, and then press Enter:

```
DELETE FROM MovieTypes
WHERE MTypeID='mt17';
```

   You should receive a response saying that your statement executed successfully, affecting one row.

How It Works

The first `DELETE` statement attempts to remove one row from the MovieTypes table, as shown in the following statement:

```
DELETE FROM MovieTypes
WHERE MTypeID='mt18';
```

The `WHERE` clause in this statement specifies that any row with an MTypeID value of mt18 should be removed from the table. Because no rows contain an MTypeID value of mt18, no rows are removed. As a result, you executed the following statement:

```
DELETE FROM MovieTypes
WHERE MTypeID='mt17';
```

This time, you specified the MTypeID value of mt17 in the `WHERE` clause. Because the MovieTypes table contains a row with an MTypeID value of mt17, that row was removed; however, no other rows were removed. Only one row in the table could contain the MTypeID value of mt17 because the MTypeID column is the primary key, so all values in that column are unique.

Deleting Data from Joined Tables

In addition to being able to delete data from an individual table that is not joined to another table, you can delete data from a joined table. The INSERT statement provides two alternatives for deleting data from a joined table: the *<from join delete>* method and the *<using join delete>* method. The *<from join delete>* method refers to the fact that the joined tables are specified in the FROM clause. The *<using join delete>* method refers to the fact that the joined tables are specified in the USING clause.

> *Generally, using a* DELETE *statement to remove data from joined tables is not recommended for* InnoDB *tables. Instead, you should rely on the* ON DELETE *and* ON CASCADE *options specified in the foreign key constraints of the table definitions.*

Using the <from join delete> Alternative of the DELETE Statement

The *<from join delete>* alternative of the DELETE statement is similar to the *<single table delete>* alternative in that it contains a FROM clause and an optional WHERE clause. As the following syntax shows, there are also a number of differences:

```
<from join delete>::=
<table name>[.*] [{, <table name>[.*]}...]
FROM <table name> [{, <table name>}...]
[WHERE <where definition>]
```

One thing that you might notice is that the syntax does not include an ORDER BY clause or a LIMIT clause. In addition, you can specify multiple tables in the DELETE clause *and* in the FROM clause. Tables specified in the DELETE clause are the tables from which data will be removed. Tables specified in the FROM clause are those tables that participate in the join. You specify tables in two separate places because this structure allows you to create a join that contains more tables than the number of tables from which data is actually deleted.

Notice that a period and an asterisk follow the table name. The period and asterisk are optional. MySQL supports their use to provide compatibility with Microsoft Access. They indicate that every column in the specified table will be included. Normally you do not need to include the period and asterisk.

Now take a look at an example of a DELETE statement that deletes data from a joined table. Earlier in the chapter, you saw two tables that demonstrated how the UPDATE statement works: the Books table and the Orders table. Suppose you use the following INSERT statements to add a row to each table:

```
INSERT INTO Books VALUES (107, 'Where I\'m Calling From', 3);
INSERT INTO Orders VALUES (1021, 107, 1, '2003-06-14 14:39:00');
```

Now suppose that you want to delete from the Orders table any order for the book title *Where I'm Calling From.* To delete these Orders, you would use the following DELETE statement:

```
DELETE Orders.*
FROM Books, Orders
WHERE Books.BookID=Orders.BookID
   AND Books.BookName='Where I\'m Calling From';
```

The DELETE clause of this statement specifies the Orders table as the table from which data is deleted. The FROM clause in the DELETE statement then goes on to specify the tables to include in the join,

which in this case are the Books and Orders tables. The `WHERE` clause, as you saw with the `UPDATE` statement, is where the join is actually defined. The first condition specified in the clause actually defines the join: `Books.BookID=Orders.BookID`. The `WHERE` clause also includes a second condition — `Books.BookName='Where I\'m Calling From'` — which indicates that only orders for this particular book should be deleted. In other words, for a row to be deleted from the Orders table, the BookID value in the Orders table must equal a BookID value in the Books table *and* the BookName value in the Books table must be *Where I'm Calling From*. When a row matches these conditions, it is deleted from the table.

In the following exercise, you attempt to delete two rows from the DVDs table in the DVDRentals database. To delete these rows, you specify a join condition in your `DELETE` statements.

Try It Out — Using the <*from join delete*> Alternative to Delete Data from the DVDRentals Database

The following steps describe the process you should follow to delete the data from the DVDs table:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To delete a record in the DVDs table, type the following `DELETE` statement at the mysql command prompt, and then press Enter:

```
DELETE DVDs
FROM DVDs, Studios
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='New Line Cinema';
```

   You should receive a response saying that your statement executed successfully but that no rows were affected.

3. To delete another record in the DVDs table, type the following `DELETE` statement at the mysql command prompt, and then press Enter:

```
DELETE DVDs
FROM DVDs, Studios
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='Metro-Goldwyn-Mayer';
```

   You should receive a response saying that your statement executed successfully, affecting one row.

How It Works

This exercise had you use two `DELETE` statements to try to delete two rows from the DVDs table. The statements were identical except for the StudDescrip value that you specified. In the first of the two statements, you specified a StudDescrip value of New Line Cinema, as shown in the following statement:

```
DELETE DVDs
FROM DVDs, Studios
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='New Line Cinema';
```

As the statement shows, the DELETE clause specifies that data should be deleted from the DVDs table. The FROM clause specifies that the DVDs table and the Studios table participate in the join. The join is then defined in the WHERE clause, which specifies that the StudID values in the DVDs table should be matched to the StudID values in the Studios table. In addition, the WHERE clause includes a second condition that specifies that the StudDescrip value in the Studios table must be New Line Cinema. Because New Line Cinema is associated with the StudID value of s109, any rows in the DVDs table that have a StudID value of s109 are deleted. When you executed this statement, no rows were deleted from the DVDs table because no rows met the criteria defined in the WHERE clause.

The next DELETE statement you ran did delete a row because you specified Metro-Goldwyn-Mayer as the StudDescrip value, rather than New Line Cinema. Because Metro-Goldwyn-Mayer is associated with the StudID value of s108 and because the DVDs table included a row that had a StudID value of s108, that row was deleted. For example, you might use a statement such as this if you want to discontinue renting all the current DVDs released by New Line Cinema, perhaps because you plan to sell the DVDs and replace them with new ones.

> *In this exercise, you used a* DELETE *statement to remove data from joined* InnoDB *tables, even though this method is normally not recommended. The exercise had you perform these deletions so that you could try out how these types of* DELETE *statements work.*

Using the <using join delete> Alternative of the DELETE Statement

The *<using join delete>* alternative of the DELETE statement is very similar to the *<from join delete>* alternative, as shown in the following syntax:

```
<using join delete>::=
FROM <table name>[.*] [{, <table name>[.*]}...]
USING <table name> [{, <table name>}...]
[WHERE <where definition>]
```

The primary differences between the *<using join delete>* alternative and the *<from join delete>* alternative are that, in the *<using join delete>* alternative, the tables from which data is deleted are specified in the FROM clause, as opposed to the DELETE clause, and the tables that are to be joined are specified in the USING clause, as opposed to the FROM clause. All other aspects of the two alternatives, however, are the same.

For example, suppose you were to rewrite the last example DELETE statement by using the *<using join delete>* alternative. The new statement would include the Books table in the FROM clause and the Books and Orders tables in the USING clause, as shown in the following example:

```
DELETE FROM Orders
USING Books, Orders
WHERE Books.BookID=Orders.BookID
   AND Books.BookName='Where I\'m Calling From';
```

As you can see in this statement, the DELETE clause doesn't include the name of the table, and the WHERE clause is identical to what you used in the DELETE statement created by using the *<from join delete>* alternative.

In the following exercise, you create two DELETE statements that use the *<using join delete>* alternative. The statements attempt to delete data from the DVDs table.

Try It Out — Using the *<using join delete>* Alternative to Delete Data from the DVDRentals Database

The following steps walk you through the process of using the *<using join delete>* alternative:

1. Open the mysql client utility, type the following command, and press Enter:

```
use DVDRentals
```

   You should receive a message indicating that you switched to the DVDRentals database.

2. To delete the first record in the DVDs table, type the following DELETE statement at the mysql command prompt, and then press Enter:

```
DELETE FROM DVDs
USING DVDs, Studios
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='New Line Cinema';
```

   You should receive a response saying that your statement executed successfully but that no rows were affected.

3. To delete the next record in the DVDs table, type the following DELETE statement at the mysql command prompt, and then press Enter:

```
DELETE FROM DVDs
USING DVDs, Studios
WHERE DVDs.StudID=Studios.StudID
   AND Studios.StudDescrip='Metro-Goldwyn-Mayer';
```

   You should receive a response saying that your statement executed successfully but that no rows were affected.

4. To delete the records in the Studios table, type the following DELETE statement at the mysql command prompt, and then press Enter:

```
DELETE FROM Studios
WHERE StudID='s108' OR StudID='s109';
```

   You should receive a response saying that your statement executed successfully, affecting two rows.

How It Works

In this exercise, you created two DELETE statements that were identical except for the StudDescrip value that you specified. The statements were intended to delete data from the DVDs table. You then used a DELETE statement to delete two rows from the Studios table. In the first DELETE statement, you specified a StudDescrip value of New Line Cinema, as shown in the following statement:

```
DELETE FROM DVDs
USING DVDs, Studios
WHERE DVDs.StudID=Studios.StudID
```

```
    AND Studios.StudDescrip='New Line Cinema';
```

The FROM clause in this statement specifies the DVDs table, which is the table from which data is removed. The USING clause specifies the DVDs and Studios table. These are the tables to be joined. The WHERE clause specifies the join condition, and a second condition that specifies that the StudDescrip value must equal New Line Cinema. The second DELETE statement is the same except that it specifies that the StudDescrip values should be Metro-Goldwyn-Mayer.

When you executed the first two DELETE statements, they were executed successfully; however, neither of them deleted any rows in the DVDs table because no rows existed that met the WHERE clause conditions specified in either statement. Even so, the exercise still demonstrated how to create a DELETE statement that uses the *<using join delete>* alternative.

After you executed the first two DELETE statements, you created the following DELETE statement to remove rows from the Studios table:

```
DELETE FROM Studios
WHERE StudID='s108' OR StudID='s109';
```

The statement uses the *<single table delete>* alternative and contains a WHERE clause that contains two conditions. The first condition specifies that the StudID value must equal s108. The second condition specifies that the StudID value should be s109. Because the two conditions are connected by an OR operator, rather than an AND operator, either condition can be met in a row. As a result, any row that contains a StudID value of s108 *or* s109 is deleted from the table. In this case, two rows were deleted.

> *As with the last Try It Out section, in this exercise, you used a* DELETE *statement to remove data from joined* InnoDB *tables, even though this method is normally not recommended. The exercise had you perform these deletions so that you could try out how these types of* DELETE *statements work.*

Using a TRUNCATE Statement to Delete Data

The final statement that this chapter covers is the TRUNCATE statement. The TRUNCATE statement removes all rows from a table. You cannot qualify this statement in any way. Any rows that exist are deleted from the target table. The following syntax describes how to create a TRUNCATE statement:

```
TRUNCATE [TABLE] <table name>
```

As you can see, you need to specify the TRUNCATE keyword and the table name. You can also specify the TABLE keyword, but it has no effect on how the table works. For example, the following TRUNCATE statement removes all data from the Orders table:

```
TRUNCATE TABLE Orders;
```

The statement includes TRUNCATE, the optional keyword TABLE, and the name of the table. If any rows exist in the table when you execute the statement, those rows are removed. Executing this statement has basically the same effect as issuing the following DELETE statement:

```
DELETE FROM Orders;
```

The most important difference between the TRUNCATE statement and the DELETE statement is that the TRUNCATE statement is not transaction safe. A *transaction* is a set of one or more SQL statements that perform a set of related actions. The statements are grouped together and treated as a single unit whose success or failure depends on the successful execution of each statement in the transaction. You learn more about transactions in Chapter 12.

Another difference between the TRUNCATE statement and the DELETE statement is that the TRUNCATE statement starts the AUTO_INCREMENT count over again, unlike the DELETE statement. TRUNCATE is generally faster than using a DELETE statement as well.

Summary

One of the most important functions of any RDBMS is to support your ability to manipulate data. To this end, you must be able to insert data in a table, modify that data, and then delete whatever data you no longer want to store. To support your ability to perform these operations, MySQL includes a number of SQL statements that allow you to insert, update, and delete data. This chapter provided you with the information you need to create these statements so that you can effectively carry out these operations. You learned how the syntax for each statement is defined and the components that make up those statements. Specifically, this chapter provided you with the information necessary to perform the following tasks:

- Use INSERT statements to add individual and multiple rows of data into tables
- Use REPLACE statements to add individual and multiple rows of data into tables
- Use UPDATE statements to modify data in single tables and joined tables
- Use DELETE statements to remove data from single tables and joined tables
- Use TRUNCATE statements to remove all data from a table

When discussing each of these statements, this chapter introduced you to a number of elements that helped to define many of the statement components. For example, the chapter provided examples of expressions, operators, functions, joins, and various statement clauses that played critical roles in defining the different types of statements. In subsequent chapters, you learn more about each of these components, which allows you to create even more robust data manipulation statements so that you can more effectively manage the data in your database and create applications that efficiently add and manipulate data.

Exercises

The following exercises help you become better acquainted with the material covered in this chapter. Be sure to work through each exercise carefully. To view the answers to these questions, see Appendix A.

1. You plan to insert data in the Books table. The table is defined with the following CREATE TABLE statement:

```
CREATE TABLE Books
(
   BookID SMALLINT NOT NULL PRIMARY KEY,
   BookName VARCHAR(50) NOT NULL
);
```

Use the *<values option>* alternative of the `INSERT` statement to add one row to the Books table. The value for the BookID column is 1001, and the name of the book is *One Hundred Years of Solitude*. Which SQL statement should you use?

2. You decide that you want to use the *<set option>* alternative of the `REPLACE` statement to add the same values to the Books table that you added in Exercise 1. Which SQL statement should you use?

3. You plan to update data in the CDs table, which is shown in the following table definition:

```
CREATE TABLE CDs
(
   CDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   CDName VARCHAR(50) NOT NULL,
   CDQuantity SMALLINT NOT NULL
);
```

The following `INSERT` statement was used to add data to the CDs table:

```
INSERT INTO CDs (CDName, CDQuantity)
VALUES ('Mule Variations', 10),
('Short Sharp Shocked', 3),
('The Bonnie Raitt Collection', 7);
```

You want to increase the CDQuantity value of every row by 3. Which SQL statement should you use?

4. You now want to increase the CDQuantity value in the CDs table by another 3, but this time, the increase should apply only to the CD named *Mule Variations*. Which SQL statement should you use?

5. You decide to delete a row from the CDs table. You want to remove the row that contains a CDID value of 1. Which SQL statement should you use?