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?