10

Accessing Data in Multiple Tables

In earlier chapters, you learned how to use SELECT statements to retrieve data from a database. As you recall, MySQL supports a number of options that allow you to create statements that are as precise as you need them to be. You can retrieve specific rows and columns, group and summarize data, or use expressions that include literal values, operators, functions, and column names. In learning about these options, most of the examples that you looked at retrieved data from only one table. MySQL also allows you to retrieve data from multiple tables and then produce one result set, as you would see when retrieving data from a single table. In fact, you can also access multiple tables from within UPDATE and DELETE statements.

MySQL supports several methods that you can use to access multiple tables in a single SQL statement. The first of these is to create a join in the statement that defines the tables to be linked together. Another method that you can use is to embed a subquery in your statement so that you can use the data returned by the subquery in the main SQL statement. In addition, you can create a union that joins together two SELECT statements in order to produce a result set that contains data retrieved by both statements. In this chapter, you learn about all three methods for accessing data in multiple tables. This chapter covers the following topics:

- Using full and outer joins in SELECT, UPDATE, and DELETE statements that link together two or more tables
- Adding subqueries to your SELECT, UPDATE, and DELETE statements that retrieve data that can be used by those statements
- Create unions that join together two SELECT statements

Creating Joins in Your SQL Statements

In a normalized database, groups of data are stored in individual tables, and relationships are established between those tables to link related data. As a result, often when creating SELECT, UPDATE, or DELETE statements, you want to be able to access data in different tables to carry out an operation affected by those relationships.

To support the capability to access data in multiple tables, MySQL allows you to create joins in a statement that define how data is accessed in multiple tables. A *join* is a condition defined in a SELECT, UPDATE, or DELETE statement that links together two or more tables. In this section, you learn how to create joins in each of these types of statements. Although much of the discussion focuses on creating joins in a SELECT statement, which is where you will most commonly use joins, many of the elements used in a SELECT join are the same elements used for UPDATE and DELETE joins.

Joining Tables in a SELECT Statement

As you're creating your SELECT statements for your applications, you may want to create statements that return data stored in different tables. The result set, though, cannot contain data that appears arbitrary in nature. In other words, the result set must be displayed in a way that suggests that the data could have been retrieved from one table. The data must be integrated and logical, despite the fact that it might come from different tables.

To achieve this integration, MySQL allows you to add joins to your SELECT statements that link together two or more tables. To illustrate how to add joins to a SELECT statement, return to the SELECT statement syntax that you first saw in Chapter 7:

```
<select statement>::=
SELECT
[<select option> [<select option>...]]
{* | <select list>}
[
   FROM {<table reference> | <join definition>}
   [WHERE <expression> [{<operator> <expression>}...]]
   [GROUP BY <group by definition>]
   [HAVING <expression> [{<operator> <expression>}...]]
   [ORDER BY <order by definition>]
   [LIMIT [<offset>,] <row count>]
]

<join definition>::=
{<table reference>, <table reference> [{, <table reference>}...]}
| {<table reference> [INNER | CROSS ] JOIN <table reference> [<join condition>]}
| {<table reference> STRAIGHT_JOIN <table reference>}
| {<table reference> LEFT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> RIGHT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> NATURAL [{LEFT | RIGHT} [OUTER]] JOIN <table reference>}

<table reference>::=
<table name> [[AS] <alias>]
[{USE | IGNORE | FORCE} INDEX <index name> [{, <index name>}...]]

<join condition>::=
ON <expression> [{<operator> <expression>}...]
| USING (<column> [{, <column>}...])
```

The syntax is not presented in its entirety and contains only those elements that are relevant to defining a join in a SELECT statement. (Refer to Chapter 7 for more information about the SELECT statement.) In addition, the syntax includes elements that you have not seen before. These elements are based on the

FROM clause, which has been modified from the original definition. As you can see, the FROM clause syntax is now as follows:

```
FROM {<table reference> | <join definition>}
```

When you originally learned about the SELECT statement syntax in Chapter 7, the clause was quite different:

```
FROM <table reference> [{, <table reference>}...]
```

Originally, the clause included only the <table reference> placeholders, and there was no mention of the <join definition> placeholder. This was done for the sake of brevity and to avoid presenting too much information at one time. As a result, the original syntax suggested that the FROM clause could include only one or more table references. Another way to describe the FROM clause is by also including the <join definition> placeholder, which defines how to add joins to a SELECT statement.

As you can see from the updated syntax of the FROM clause, the clause can include either a table reference or a join definition. You've already seen numerous examples of SELECT statements that use the <table reference> format. These are the SELECT statements that include only one table name in the FROM clause. If you plan to reference multiple tables in your FROM clause, you are defining some type of join, in which case the <join definition> placeholder applies.

The <join definition> placeholder refers to a number of different types of joins, as the following syntax shows:

```
<join definition>::=
{<table reference>, <table reference> [{, <table reference>}...]}
| {<table reference> [INNER | CROSS ] JOIN <table reference> [<join condition>]}
| {<table reference> STRAIGHT_JOIN <table reference>}
| {<table reference> LEFT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> RIGHT [OUTER] JOIN <table reference> [<join condition>]}
| {<table reference> NATURAL [{LEFT | RIGHT} [OUTER]] JOIN <table reference>}
```

The first of these is the basic join, which is made up of only table references separated by commas. The other joins are the inner and cross joins, the straight join, the left join, the right join, and the natural join. Later in this section, you learn about each type of join. For now, the most important point to know is that you can define a FROM clause with a table reference or with one of several join definitions. It's also worth noting that each join definition includes one or two <table reference> placeholders. This is the same <table reference> placeholder used when the FROM clause includes no join definition. The following syntax describes the table reference:

```
<table reference>::=
<table name> [[AS] <alias>]
[{USE | IGNORE | FORCE} INDEX <index name> [{, <index name>}...]]
```

As you recall from Chapter 7, the <table reference> placeholder refers not only to the table name, but also to the syntax used to assign an alias to that table or define index-related options. You use table references in your joins in the same way you use a table reference directly in a FROM clause. At the very least, you must provide a table name, but you can also use any of the other options.

As you move through this section, you learn how to create each type of join. To demonstrate how to create joins, the section provides a number of examples. The examples are based on three tables: Books, Authors, and AuthorBook. The table definition for the Books table is as follows:

```
CREATE TABLE Books
(
   BookID SMALLINT NOT NULL PRIMARY KEY,
   BookTitle VARCHAR(60) NOT NULL,
   Copyright YEAR NOT NULL
)
ENGINE=INNODB;
```

For the purposes of the examples, you can assume that the following `INSERT` statement populated the Books table:

```
INSERT INTO Books
VALUES (12786, 'Letters to a Young Poet', 1934),
(13331, 'Winesburg, Ohio', 1919),
(14356, 'Hell\'s Angels', 1966),
(15729, 'Black Elk Speaks', 1932),
(16284, 'Noncomformity', 1996),
(17695, 'A Confederacy of Dunces', 1980),
(19264, 'Postcards', 1992),
(19354, 'The Shipping News', 1993);
```

The Authors table is the next that you use, which is shown in the following `CREATE TABLE` statement:

```
CREATE TABLE Authors
(
   AuthID SMALLINT NOT NULL PRIMARY KEY,
   AuthFN VARCHAR(20),
   AuthMN VARCHAR(20),
   AuthLN VARCHAR(20)
)
ENGINE=INNODB;
```

The following `INSERT` statement shows the values that have been inserted in the Authors table:

```
INSERT INTO Authors
VALUES (1006, 'Hunter', 'S.', 'Thompson'),
(1007, 'Joyce', 'Carol', 'Oates'),
(1008, 'Black', NULL, 'Elk'),
(1009, 'Rainer', 'Maria', 'Rilke'),
(1010, 'John', 'Kennedy', 'Toole'),
(1011, 'John', 'G.', 'Neihardt'),
(1012, 'Annie', NULL, 'Proulx'),
(1013, 'Alan', NULL, 'Watts'),
(1014, 'Nelson', NULL, 'Algren');
```

Finally, the table definition for the AuthorBook table is as follows:

```
CREATE TABLE AuthorBook
(
   AuthID SMALLINT NOT NULL,
```

```
    BookID SMALLINT NOT NULL,
    PRIMARY KEY (AuthID, BookID),
    FOREIGN KEY (AuthID) REFERENCES Authors (AuthID),
    FOREIGN KEY (BookID) REFERENCES Books (BookID)
)
ENGINE=INNODB;
```

The following `INSERT` statement has been used to add data to the AuthorBook table:

```
INSERT INTO AuthorBook
VALUES (1006, 14356), (1008, 15729), (1009, 12786), (1010, 17695),
(1011, 15729), (1012, 19264), (1012, 19354), (1014, 16284);
```

Now that you've seen the table definitions for the Books, Authors, and AuthorBook tables, take a look at Figure 10-1, which illustrates how these three tables are related to one another. Notice that a one-to-many relationship exists between the Books and AuthorBook tables, and one exists between the Authors and AuthorBook tables. What this implies is that each book may have been written by one or more authors and that each author may have written one or more books.

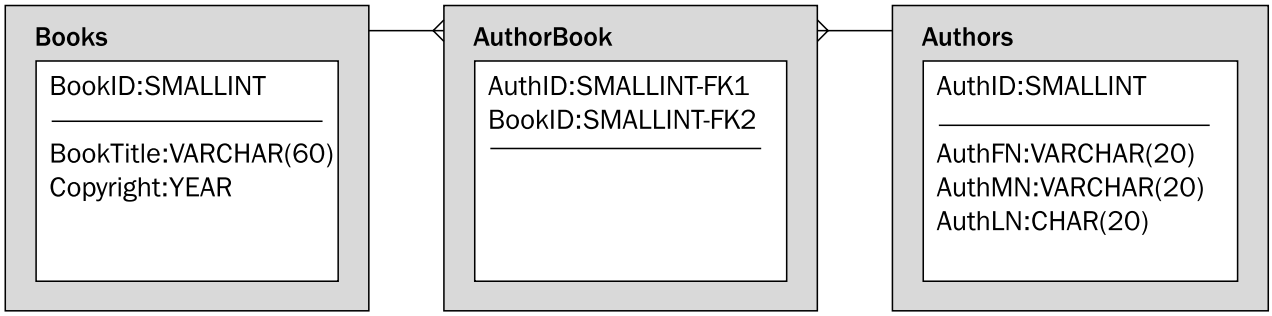

Figure 10-1

As you learn more about each type of join, you might want to refer to these table definitions and the illustration as a reference. In the meantime, take a look at the different types of joins supported by MySQL. You can divide these joins into two broad categories: full joins and outer joins. Full joins and outer joins are distinguished from one another in the way that they match rows in the tables that are being joined. In a full join, the `SELECT` statement returns only rows that match the join condition. In an outer join, the rows that match the join condition are returned as well as additional rows. As you work your way through the chapter, you better understand how the two types of joins differ from one another. Keep in mind, however, that the term "full join" is used here only as a way to group together similar types of nonouter joins. The term used to describe these types of joins varies greatly in different documentation and in different RDBMS products. In some cases, the two types of joins are differentiated by referring to inner joins and outer joins. Because one type of full join that MySQL supports is referred to as an inner join, categorizing the joins in this way can lead to confusion. The important point to remember is that the terms "full join" and "outer join" are based on the results returned by the `SELECT` statement, with full joins being the more common of the two.

Creating Full Joins

MySQL supports several types of full joins: the basic join, the inner and cross joins, and the straight join. Of these, the basic join is the one most commonly used, so that is where the discussion begins.

Creating Basic Joins

In some of the examples provided earlier in the book, you saw how to use basic joins. As you recall, you simply specified a `FROM` clause, along with multiple table names, separated by commas. The following syntax shows how a basic join is defined in the `FROM` clause:

```
<table reference>, <table reference> [{, <table reference>}...]
```

As you can see, the basic join must include at least two table references, but you can add as many table references as necessary. Just remember that commas must separate the table references. For example, the following `SELECT` statement creates a join on the Books and AuthorBook tables:

```
SELECT BookTitle, Copyright, AuthID
FROM Books, AuthorBook
ORDER BY BookTitle;
```

This statement is very similar to many of the `SELECT` statements that you've already seen, except that the `FROM` clause includes two tables: Books and AuthorBook. Because the `FROM` clause references the Books and the AuthorBook tables, these are the tables that are joined together.

Now take a look at the `SELECT` clause. The clause references three columns: BookTitle, Copyright, and AuthID. The first two columns, BookTitle and Copyright, are in the Books table, and the third column, AuthID, is in the Authors table.

The only other clause in the `SELECT` statement is the `ORDER BY` clause, which specifies that the query results be sorted according to the values in the BookTitle column.

In the previous example `SELECT` statement, a join is defined on two tables; however, the join is not qualified in any other way. (The columns referenced in the `SELECT` clause or the column referenced in the `ORDER BY` clause do not affect how the join itself is defined.) When a basic join is not qualified in any way, it returns a result set that matches every row in one table to every row in the joined table. This type of result set is referred to as a *Cartesian product.* For example, the previous `SELECT` statement returns the following Cartesian product:

```
+-------------------------+-----------+---------+
| BookTitle               | Copyright | AuthID |
+-------------------------+-----------+---------+
| A Confederacy of Dunces |      1980 |   1008 |
| A Confederacy of Dunces |      1980 |   1009 |
| A Confederacy of Dunces |      1980 |   1010 |
| A Confederacy of Dunces |      1980 |   1011 |
| A Confederacy of Dunces |      1980 |   1012 |
| A Confederacy of Dunces |      1980 |   1012 |
| A Confederacy of Dunces |      1980 |   1014 |
| A Confederacy of Dunces |      1980 |   1006 |
| Black Elk Speaks        |      1932 |   1012 |
| Black Elk Speaks        |      1932 |   1012 |
| Black Elk Speaks        |      1932 |   1014 |
| Black Elk Speaks        |      1932 |   1006 |
| Black Elk Speaks        |      1932 |   1008 |
| Black Elk Speaks        |      1932 |   1009 |
| Black Elk Speaks        |      1932 |   1010 |
```

```
| Black Elk Speaks         |      1932 |   1011 |
| Hell's Angels            |      1966 |   1014 |
| Hell's Angels            |      1966 |   1006 |
| Hell's Angels            |      1966 |   1008 |
```

The results shown here are only part of the rows returned by the `SELECT` statement. The statement returns 64 rows in all. This is based on the fact that each table named in the `FROM` clause contains 8 rows. If each row in the Books table is matched to each row in the AuthorBook table, there are 8 AuthorBook rows for each Books row, or a total of 64 rows. For example, in the previous result set, you can see that the first 8 rows have a BookTitle value of A Confederacy of Dunces. Each of these rows is matched with 1 row from the AuthorBook table. Because the `SELECT` clause specifies that the AuthID value from the AuthorBook table should be displayed, that value from each row is displayed.

Using a join to return a Cartesian product is normally not very useful, and it could potentially return a great many rows. For example, if you joined 3 tables that each included 100 rows (relatively small tables in today's world of databases), the Cartesian product returned by joining these tables would be made up of 1 million rows (100 x 100 x 100). As a result, most joins must be qualified in some way to limit the number of rows returned and to ensure that the result set contains only data that is both useful and manageable.

When using a basic join to retrieve data from multiple tables, you can qualify the join by adding the necessary conditions in the `WHERE` clause. For example, the following `SELECT` statement is similar to the last, except for the addition of a `WHERE` clause:

```
SELECT BookTitle, Copyright, AuthID
FROM Books AS b, AuthorBook AS ab
WHERE b.BookID=ab.BookID
ORDER BY BookTitle;
```

The `WHERE` clause includes a condition that defines the join's limitations. In this case, the values in the BookID column in the Books table (`b.BookID`) are matched to the values in the BookID column of the AuthorBook table (`ab.BookID`). As a result, the only rows that are returned are those in which the BookID values in both tables are equal.

Notice that the Books and AuthorBook tables have been assigned alias names, `b` and `ab`, respectively. Aliases are assigned to table names to make referencing those tables easier in other parts of the statement. For example, each BookID reference in the `WHERE` clause is qualified with the alias for the table name. The aliases are assigned to the table names in the `FROM` clause. Thereafter, any references to the tables are through the use of the aliases. If there is no chance of confusing a column name with another column, you do not need to qualify the column name. If the same column name is used in more than one table, the name must be qualified so that MySQL knows which column is being referenced.

By adding the `WHERE` clause to the statement, the number of rows returned by the statement is drastically reduced, as shown in the following result set:

```
+--------------------------+-----------+--------+
| BookTitle                | Copyright | AuthID |
+--------------------------+-----------+--------+
| A Confederacy of Dunces  |      1980 |   1010 |
| Black Elk Speaks         |      1932 |   1008 |
| Black Elk Speaks         |      1932 |   1011 |
| Hell's Angels            |      1966 |   1006 |
| Letters to a Young Poet  |      1934 |   1009 |
```

```
| Noncomformity           |      1996 |   1014 |
| Postcards               |      1992 |   1012 |
| The Shipping News       |      1993 |   1012 |
+-------------------------+-----------+--------+
8 rows in set (0.00 sec)
```

As you can see, this is a very different result from the Cartesian product returned by the first example you looked at. Now only rows that have the same BookID value in both columns are returned. For example, the row in the Books table that contains a BookTitle value of A Confederacy of Dunces is matched to the row in the AuthorBook table that contains an AuthID value of 1010. The rows in both tables have the same BookID value, which is 17695.

In addition to the `WHERE` clause containing a condition that specifies the extent of the join, the `WHERE` clause can include additional logical operators and expressions that further limit the results returned by the `SELECT` statement. For example, the following `SELECT` statement returns only those rows that contain a Copyright value less than 1980:

```
SELECT BookTitle, Copyright, ab.AuthID
FROM Books AS b, AuthorBook AS ab
WHERE b.BookID=ab.BookID AND Copyright<1980
ORDER BY BookTitle;
```

As you can see, this `SELECT` statement is identical to the preceding one, except for the addition of an expression in the `WHERE` clause. Now the statement returns only those rows with matching BookID values *and* whose Copyright value is less than 1980, as shown in the following query results:

```
+-------------------------+-----------+--------+
| BookTitle               | Copyright | AuthID |
+-------------------------+-----------+--------+
| Black Elk Speaks        |      1932 |   1008 |
| Black Elk Speaks        |      1932 |   1011 |
| Hell's Angels           |      1966 |   1006 |
| Letters to a Young Poet |      1934 |   1009 |
+-------------------------+-----------+--------+
4 rows in set (0.00 sec)
```

As you can see, the statement returns only 4 rows, which is quite a difference from the 64 rows returned by the first statement. As this last example demonstrates, the more specific you can be in your `SELECT` statement, the more precise the results.

In the last few examples that you've seen, the `SELECT` statements included joins that linked two tables together. As you can see from the query results, returning only an AuthID value might not always be very useful. Wouldn't it be better to know the author's name for that book?

Fortunately, MySQL allows you to join more than two tables together. The basic statement is similar to what you've already seen; only a few supplemental elements are added to bring in the additional table, as shown in the following `SELECT` statement:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b, AuthorBook AS ab, Authors AS a
WHERE b.BookID=ab.BookID AND ab.AuthID=a.AuthID AND Copyright<1980
ORDER BY BookTitle;
```

Now the FROM clause includes three tables: Books, AuthorBook, and Authors. Each table is assigned an alias, which is then used in the WHERE clause. The WHERE clause also contains an additional element, which is a condition that specifies that the AuthID value in the AuthorBook table must match the AuthID value in the Authors table. When all the conditions in the WHERE clause are taken together, the result is that, for each row returned by the result set, the BookID value in the Books table must equal the BookID value in the AuthorBook table *and* the AuthID value in the AuthorBook table must equal the AuthID value in the Authors table *and* the Copyright value must be less than 1980, as shown in the following query results:

```
+-------------------------+-----------+--------------------+
| BookTitle               | Copyright | Author             |
+-------------------------+-----------+--------------------+
| Black Elk Speaks        |      1932 | Black Elk          |
| Black Elk Speaks        |      1932 | John G. Neihardt   |
| Hell's Angels           |      1966 | Hunter S. Thompson |
| Letters to a Young Poet |      1934 | Rainer Maria Rilke |
+-------------------------+-----------+--------------------+
4 rows in set (0.00 sec)
```

As you can see, the query results now show the name of the author rather than the AuthID value. The author's first, middle, and last names are taken from the Authors table and concatenated in one value. As a result, the query returns information that is far more useful than you saw in the preceding statements, and you no longer have to try to decipher primary key values to determine who the author is.

Creating Inner Joins and Cross Joins

When it comes to joins, MySQL provides numerous ways to accomplish the same results. For example, you can create inner joins and cross joins that produce identical result sets as those generated by the basic joins you saw in the previous section. As you recall from the Introduction, there are a number of reasons why MySQL provides different methods for achieving the same results. For example, as MySQL has tried to conform to the SQL standard, statements have evolved from one version of MySQL to the next; however, the original statement is supported for legacy systems. In addition, by supporting different ways to achieve different results, your code can be more portable in an application. For instance, one version of the SQL statement might be supported in another RDBMS, so the statement can be used to retrieve data from two different database systems. If you're creating an application that accesses only a MySQL database and that database is managed by a current version of MySQL, you're usually safe using the simplest statement available. For more complex joins, you might want to try different types of joins to determine whether one performs better than the other. Normally, though, this isn't necessary.

Now take a look at how an inner or cross join is created. The following syntax shows you how to add an inner or cross join to your SELECT statement:

```
<table reference> [INNER | CROSS ] JOIN <table reference> [<join condition>]

<join condition>::=
ON <expression> [{<operator> <expression>}...]
| USING (<column> [{, <column>}...])
```

As the syntax illustrates, you must include a table reference, the optional INNER or CROSS keyword, the JOIN keyword, the second table reference, and an optional join condition. The join condition can consist of an ON clause or a USING clause. The join condition defines the circumstances of the join. In the basic

join, the circumstances are defined in the WHERE clause, but in the inner and cross joins, this is accomplished through the join condition.

For all practical purposes, specifying the INNER or CROSS keywords produces the same results as not specifying either. (When neither the INNER nor the CROSS keywords are specified, the join is usually referred to as a cross join.) In addition, if you don't qualify the join in any way, the results produce the same Cartesian product as their basic join counterpart. For example, the following four statements produce the same results:

```
SELECT BookTitle, AuthID FROM Books, AuthorBook;
SELECT BookTitle, AuthID FROM Books JOIN AuthorBook;
SELECT BookTitle, AuthID FROM Books INNER JOIN AuthorBook;
SELECT BookTitle, AuthID FROM Books CROSS JOIN AuthorBook;
```

Each statement returns 64 rows. Each row in the Books table is matched to each row in the AuthorBook table. As a result, the join must be qualified in order to produce useful results. To qualify the inner or cross join, you can use the ON or USING clause rather than the WHERE clause. For example, the following SELECT statement uses an ON clause to qualify the join condition:

```
SELECT BookTitle, Copyright, ab.AuthID
FROM Books AS b JOIN AuthorBook AS ab
   ON b.BookID=ab.BookID
ORDER BY BookTitle;
```

As you can see, the FROM clause defines the join. Rather than separating the joined tables with a comma, the statement uses the JOIN keyword. In addition, the ON clause specifies that the BookID values in the Books and AuthorBook tables must be equal in order for a row to be returned. By defining a join condition in an ON clause, the results are limited to eight rows, as shown in the following result set:

```
+-------------------------+-----------+--------+
| BookTitle               | Copyright | AuthID |
+-------------------------+-----------+--------+
| A Confederacy of Dunces |      1980 |   1010 |
| Black Elk Speaks        |      1932 |   1008 |
| Black Elk Speaks        |      1932 |   1011 |
| Hell's Angels           |      1966 |   1006 |
| Letters to a Young Poet |      1934 |   1009 |
| Noncomformity           |      1996 |   1014 |
| Postcards               |      1992 |   1012 |
| The Shipping News       |      1993 |   1012 |
+-------------------------+-----------+--------+
8 rows in set (0.00 sec)
```

You can produce the same results by using a USING clause to qualify the join, as shown in the following statement:

```
SELECT BookTitle, Copyright, ab.AuthID
FROM Books JOIN AuthorBook AS ab
   USING (BookID)
ORDER BY BookTitle;
```

As you can see, the USING clause requires only that you specify the joined columns (in parentheses). You can use this technique only when the joined columns in each table share the same name. If you create a join that is based on more than one column, you must specify all those columns and separate them with a comma.

As is the case with the basic join, you can also further qualify an inner or full join by specifying a condition in the WHERE clause, as shown in the following statement:

```
SELECT BookTitle, Copyright, ab.AuthID
FROM Books AS b JOIN AuthorBook AS ab
   ON b.BookID=ab.BookID
WHERE Copyright<1980
ORDER BY BookTitle;
```

When defining inner or cross joins, you should specify the linked columns in the ON or USING clause and any other conditions in the WHERE clause. You can also specify the join conditions in the WHERE clause, as you do in a basic join, and not use ON or USING clauses. Your statements are generally easier to read, however, if you use ON or USING clauses.

Returning now to the preceding SELECT statement, if you execute this statement, it produces the same results as its basic join counterpart, as shown in the following result set:

```
+-------------------------+-----------+--------+
| BookTitle               | Copyright | AuthID |
+-------------------------+-----------+--------+
| Black Elk Speaks        |      1932 |   1008 |
| Black Elk Speaks        |      1932 |   1011 |
| Hell's Angels           |      1966 |   1006 |
| Letters to a Young Poet |      1934 |   1009 |
+-------------------------+-----------+--------+
4 rows in set (0.00 sec)
```

You can also create inner and cross joins on more than two tables. As with the preceding example, you should specify the necessary join conditions in an ON or USING clause. For example, the following SELECT statement includes two ON clauses:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b CROSS JOIN AuthorBook AS ab ON b.BookID=ab.BookID
   CROSS JOIN Authors AS a ON ab.AuthID=a.AuthID
WHERE Copyright<1980
ORDER BY BookTitle;
```

In this statement, the FROM clause first joins the Books table to the AuthorBook table. The join is qualified through the use of an ON clause that matches the values in the BookID column of each table. The FROM clause then goes on to join the AuthorBook table to the Authors table. The second join is qualified through the use of an ON clause that matches the values of the AuthID column of the Authors table and the AuthorBook table. Consequently, the following results are produced:

```
+-------------------------+-----------+--------------------+
| BookTitle               | Copyright | Author             |
+-------------------------+-----------+--------------------+
| Black Elk Speaks        |      1932 | Black Elk          |
| Black Elk Speaks        |      1932 | John G. Neihardt   |
| Hell's Angels           |      1966 | Hunter S. Thompson |
| Letters to a Young Poet |      1934 | Rainer Maria Rilke |
+-------------------------+-----------+--------------------+
4 rows in set (0.00 sec)
```

Finally, you can achieve the same results in the preceding statement by using two `USING` clauses, rather than `ON` clauses, as shown in the following statement:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books JOIN AuthorBook USING (BookID)
   JOIN Authors USING (AuthID)
WHERE Copyright<1980
ORDER BY BookTitle;
```

As you can see, the `USING` clause is a little simpler to create and read because you have to specify the linking columns only once. The `USING` clause is also handy because it often eliminates the need to use aliases.

Creating Straight Joins

Another type of join that you can create is the straight join, which is similar to a basic join in most respects. The primary difference is that a straight join allows you to specify that the join optimizer read the table on the left before the table on the right. (The join optimizer is a component of the MySQL server that tries to determine the best way to process a join.) You would use a straight join in those cases in which you believe that the join optimizer is not processing your `SELECT` statement as efficiently as it could. In most cases, though, you should rely on the optimizer.

To create a straight join, you can use the following syntax in your `FROM` clause:

```
<table reference> STRAIGHT_JOIN <table reference>
```

As you can see, you must specify a table reference, followed by the `STRAIGHT_JOIN` keyword, which is then followed by another table reference. For example, the following `SELECT` statement produces the same results that you saw in the previous two examples:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b STRAIGHT_JOIN AuthorBook AS ab STRAIGHT_JOIN Authors AS a
WHERE b.BookID=ab.BookID AND ab.AuthID=a.AuthID AND Copyright<1980
ORDER BY BookTitle;
```

One thing to notice about the straight join is that, like the basic join, the join is qualified in the `WHERE` clause, not in an `ON` or `USING` clause. In fact, the only difference between a straight join and a basic join, in terms of syntax, is that in a straight join you use the `STRAIGHT_JOIN` keyword to separate the table references, rather than a comma.

MySQL provides another method that you can use to create a straight join. Instead of using the straight join syntax in the FROM clause, you use the basic join syntax and you specify the STRAIGHT_JOIN table option in the SELECT clause, as shown in the following example:

```
SELECT STRAIGHT_JOIN BookTitle, Copyright,
   CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b, AuthorBook AS ab, Authors AS a
WHERE b.BookID=ab.BookID AND ab.AuthID=a.AuthID AND Copyright<1980
ORDER BY BookTitle;
```

As you can see, the table references in the FROM clause are separated by commas, which is typical of a basic join. In addition, the SELECT clause includes the STRAIGHT_JOIN select option. This statement returns the same results that you saw in the preceding three examples.

So far in this chapter, you have learned how to create SELECT statements that use full joins to retrieve data from two or more tables. In this Try It Out section you create your own SELECT statements that retrieve data from multiple tables in the DVDRentals database.

Try It Out Using Full Joins to Retrieve Data

Follow these steps to create statements that use full joins:

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. The first SELECT statement that you create uses a basic join to retrieve DVDName values from the DVDs table and MTypeDescrip values from the MovieTypes table. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip As MovieType
FROM DVDs AS d, MovieTypes AS mt
WHERE d.MTypeID=mt.MTypeID AND StatID='s2'
ORDER BY DVDName;
```

   You should receive results similar to the following:

```
+------------------------------+-----------+
| DVDName                      | MovieType |
+------------------------------+-----------+
| Amadeus                      | Drama     |
| Mash                         | Comedy    |
| The Maltese Falcon           | Drama     |
| The Rocky Horror Picture Show | Comedy    |
| What's Up, Doc?              | Comedy    |
+------------------------------+-----------+
5 rows in set (0.00 sec)
```

3. The next SELECT statement that you create is similar to the preceding one except that you also retrieve data from the Ratings table. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip As MovieType,
   CONCAT(d.RatingID, ': ', r.RatingDescrip) AS Rating
FROM DVDs AS d, MovieTypes AS mt, Ratings AS r
WHERE d.MTypeID=mt.MTypeID AND d.RatingID=r.RatingID
   AND StatID='s2'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+------------------------------+-----------+---------------------------------+
| DVDName                      | MovieType | Rating                          |
+------------------------------+-----------+---------------------------------+
| Amadeus                      | Drama     | PG: Parental guidance suggested |
| Mash                         | Comedy    | R: Under 17 requires adult      |
| The Maltese Falcon           | Drama     | NR: Not rated                   |
| The Rocky Horror Picture Show | Comedy    | NR: Not rated                   |
| What's Up, Doc?              | Comedy    | G: General audiences            |
+------------------------------+-----------+---------------------------------+
5 rows in set (0.00 sec)
```

4. Now retrieve the same data that your retrieved in the preceding SELECT statement, only this time, use a cross join. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip AS MovieType,
   CONCAT(d.RatingID, ': ', r.RatingDescrip) AS Rating
FROM MovieTypes AS mt CROSS JOIN DVDs AS d USING (MTypeID)
   CROSS JOIN Ratings AS r USING (RatingID)
WHERE StatID='s2'
ORDER BY DVDName;
```

You should receive results similar to those you received in the preceding statement.

5. In the next SELECT statement that you create, you use an inner join to retrieve customer names and the transactions IDs for those transactions that involve the DVD with a DVDID value of 4. Execute the following SQL statement at the mysql command prompt:

```
SELECT CONCAT_WS(' ', CustFN, CustMN, CustLN) AS Customer, TransID
FROM Customers INNER JOIN Orders USING (CustID)
   INNER JOIN Transactions USING (OrderID)
WHERE DVDID=4
ORDER BY CustLN;
```

You should receive results similar to the following:

```
+-------------------------+---------+
| Customer                | TransID |
+-------------------------+---------+
| Ralph Frederick Johnson |       2 |
| Peter Taylor            |      12 |
| Anne Thomas             |       8 |
| Hubert T. Weatherby     |       5 |
+-------------------------+---------+
4 rows in set (0.01 sec)
```

6. Next, use a basic join in a `SELECT` statement to join together four tables in order to display the names of movie participants, the movies that they participated in, and the roles they played in those movies. Execute the following SQL statement at the mysql command prompt:

```
SELECT CONCAT_WS(' ', PartFN, PartMN, PartLN) AS Participant, DVDName,
   RoleDescrip AS Role
FROM DVDs AS d, DVDParticipant AS dp, Participants AS p, Roles AS r
WHERE d.DVDID=dp.DVDID AND p.PartID=dp.PartID AND r.RoleID=dp.RoleID
ORDER BY PartLN;
```

You should receive results similar to the following:

```
+------------------+--------------------+--------------+
| Participant      | DVDName            | Role         |
+------------------+--------------------+--------------+
| John Barry       | Out of Africa      | Composer     |
| Irving Berlin    | White Christmas    | Composer     |
| Humphrey Bogart  | The Maltese Falcon | Actor        |
| Henry Buck       | What's Up, Doc?    | Screenwriter |
| Rosemary Clooney | White Christmas    | Actor        |
| Bing Crosby      | White Christmas    | Actor        |
| Michael Curtiz   | White Christmas    | Director     |
| Danny Kaye       | White Christmas    | Actor        |
| Sydney Pollack   | Out of Africa      | Producer     |
| Sydney Pollack   | Out of Africa      | Director     |
| Robert Redford   | Out of Africa      | Actor        |
| Meryl Streep     | Out of Africa      | Actor        |
+------------------+--------------------+--------------+
12 rows in set (0.02 sec)
```

How It Works

In this exercise, you created five `SELECT` statements that each use a different type of join to integrate data from more than one table in a single result set. In this first statement, you defined a basic join that integrated data from the DVDs table and the MovieTypes table:

```
SELECT DVDName, MTypeDescrip As MovieType
FROM DVDs AS d, MovieTypes AS mt
WHERE d.MTypeID=mt.MTypeID AND StatID='s2'
ORDER BY DVDName;
```

The join is defined in the `FROM` clause. By including more than one table in the clause, you're specifying that a join be created. Because this is a basic join, the join is qualified by adding a condition to the `WHERE` clause that specifies that the MTypeID values in both tables be equal. The `WHERE` clause also contains a second condition that specifies that the StatID value must be s2. As a result, any row returned by this statement must have equal values in the MTypeID columns of both tables *and* the StatID value in the DVDs table must equal s2.

The next `SELECT` statement is similar to the preceding one except that you added a third table to the statement:

```
SELECT DVDName, MTypeDescrip As MovieType,
   CONCAT(d.RatingID, ': ', r.RatingDescrip) AS Rating
FROM DVDs AS d, MovieTypes AS mt, Ratings AS r
```

```
WHERE d.MTypeID=mt.MTypeID AND d.RatingID=r.RatingID
   AND StatID='s2'
ORDER BY DVDName;
```

The FROM clause now includes the Ratings table in addition to the DVDs and MovieTypes tables. In addition, you added a third condition to the WHERE clause that specifies that the RatingID values in the DVDs table and the Ratings table must be equal. As a result, any row returned by this statement must have equal values in the MTypeID columns of the DVDs and MovieTypes tables *and* values in the RatingID columns of the DVDs and Ratings tables *and* the StatID value in the DVDs table must equal s2. In addition, the updated SELECT statement also includes a third column in the result set that concatenates the RatingID value and the RatingDescrip value for each returned row.

You then created a SELECT statement that retrieved the same results as the preceding statement. The next statement, however, uses a cross join rather than a basic join:

```
SELECT DVDName, MTypeDescrip AS MovieType,
   CONCAT(d.RatingID, ': ', r.RatingDescrip) AS Rating
FROM MovieTypes AS mt CROSS JOIN DVDs AS d USING (MTypeID)
   CROSS JOIN Ratings AS r USING (RatingID)
WHERE StatID='s2'
ORDER BY DVDName;
```

As the statement shows, the FROM clause now includes the join definitions and the USING clauses. The MovieTypes table is joined to the DVDs table, and that join is qualified through the USING clause, which specifies the MTypeID column. The Orders DVDs table is then joined to the Ratings table, and that join is qualified through the USING clause, which specifies the RatingID column.

In the next SELECT statement that you created, you defined an inner join on the Customers, Orders, and Transactions table:

```
SELECT CONCAT_WS(' ', CustFN, CustMN, CustLN) AS Customer, TransID
FROM Customers INNER JOIN Orders USING (CustID)
   INNER JOIN Transactions USING (OrderID)
WHERE DVDID=4
ORDER BY TransID;
```

The Customers table is joined to the Orders table, and that join is qualified through the USING clause, which specifies the CustID column. The Orders table is then joined to the Transactions table, and that join is qualified through the USING clause, which specifies the Order ID column. The statement returns the customer name (concatenated) from the Customers table and the TransID value from the Transactions table. Only rows that have a DVDID value of 4 are returned, as specified by the WHERE clause.

The final SELECT statement that you created joined the DVDs, DVDParticipant, Participants, and Roles table:

```
SELECT CONCAT_WS(' ', PartFN, PartMN, PartLN) AS Participant, DVDName,
   RoleDescrip AS Role
FROM DVDs AS d, DVDParticipant AS dp, Participants AS p, Roles AS r
WHERE d.DVDID=dp.DVDID AND p.PartID=dp.PartID AND r.RoleID=dp.RoleID
ORDER BY PartLN;
```

The statement creates a basic join and defines the join conditions in the `WHERE` clause, which states that the DVDID values in the DVDs and DVDParticipant tables must be equal, the PartID values in the Participants and DVDParticipant tables must be equal, and the RoleID values in the Roles and DVDParticipant tables must be equal. The result set includes a list of the participants (from the Participants table), the names of the movies in which they participated (from the DVDs table), and the roles they played (from the Roles table). The link among these three tables is defined through the DVDParticipant table, which is why all tables must join to that table.

As you have seen, MySQL supports a number of full joins. There might be times when you want to display results that include rows in addition to those that match the join condition. To include additional rows in your result sets, you can use the outer join.

Creating Outer Joins

Except for the joins that returned Cartesian products, the joins that you have seen so far have been those that returned only rows that contained matching values in the specified columns. Suppose that you want to include details that do not fit these conditions, yet at the same time, you don't want to return large result sets of unnecessary information. For example, in the statements that you looked at when learning about full joins, you saw joins that were based on the Books, AuthorBook, and Authors tables (shown in Figure 10-1). Some of the joins that were created allowed you to match authors to books so that you were able to view books and their authors together. What if the Books table contains books for which no authors are listed in the Authors table? Or authors are listed in the Authors table that do not have books listed in the Books table?

MySQL allows you to view these authors or books by using outer joins. You can create one of two types of outer joins: a left join or a right join. A left join pulls all values from the table on the left, and a right join pulls all values from the table listed on the right. To help illustrate this point, Figure 10-2 shows a Venn diagram that represents the left outer join. The circle on the left represents the Books table, and the circle on the right represents the Authors table. The Venn diagram illustrates what will happen when these tables are joined together. The values on the left side of the join are taken from the Books table, and the values on the right side of the join are taken from the Authors table. In a full join, only the values that are matched by the join are returned. These are the values that appear in the section where the two circles intersect. As you can see, each title is matched with one or more authors.

Figure 10-2 shows something else. The book *Winesburg, Ohio* cannot be matched with an author because no author exists in the Authors table for that book. In a full join, this book is not included in the results set. Because this is a left outer join, the book is still returned, but it is matched with a value of `NULL`.

If this were a right outer join, any authors that do not match a book in the Books table are paired with `NULL` values, as is the case of the books for which no authors are listed. The following sections examine each type of outer join to better illustrate how they work.

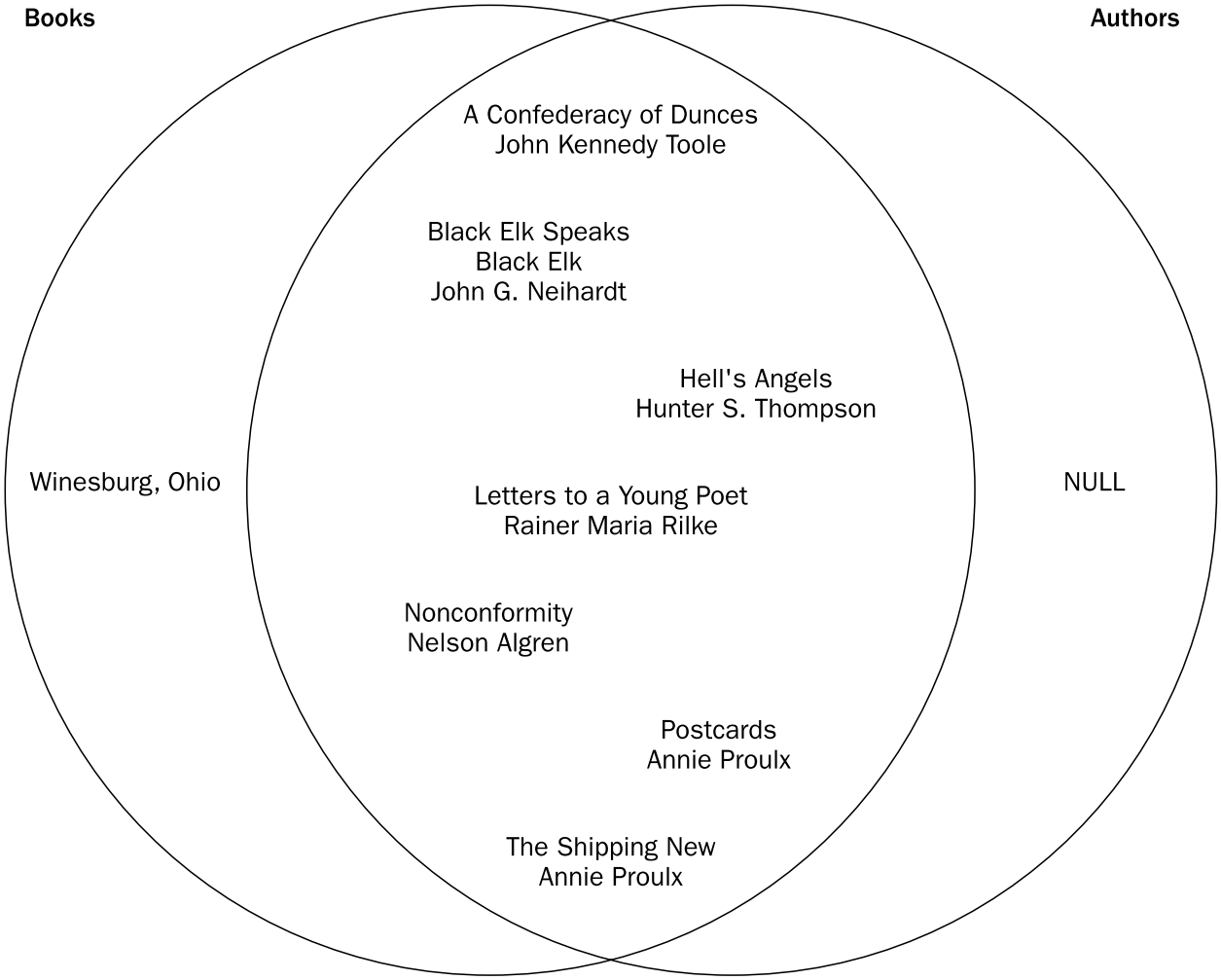

Figure 10-2

Creating Left Joins

The syntax for a left outer join is similar to an inner or cross join, except that you must include the `LEFT` keyword, as shown in the following syntax:

```
<table reference> LEFT [OUTER] JOIN <table reference> [<join condition>]

<join condition>::=
ON <expression> [{<operator> <expression>}...]
| USING (<column> [{, <column>}...])
```

As you can see, you first specify a table reference, followed by the `LEFT` keyword, the optional `OUTER` keyword, and then the `JOIN` keyword. This is all followed by another table reference and an optional join condition, which can be an `ON` clause or a `USING` clause. For example, the following `SELECT` statement defines a left outer join on the Books and AuthorBook tables:

```
SELECT BookTitle, Copyright, AuthID
FROM Books AS b LEFT JOIN AuthorBook AS ab
   ON b.BookID=ab.BookID
ORDER BY BookTitle;
```

As you can see, the `FROM` clause first references the Books table and then references the AuthorBook table. As a result, all results are returned from the Books table (the left table), and only the results that meet the condition specified in the `ON` clause are returned from the AuthorBook table (the right table), as shown in the following result set:

```
+--------------------------+-----------+--------+
| BookTitle                | Copyright | AuthID |
+--------------------------+-----------+--------+
| A Confederacy of Dunces |      1980 |   1010 |
| Black Elk Speaks         |      1932 |   1008 |
| Black Elk Speaks         |      1932 |   1011 |
| Hell's Angels            |      1966 |   1006 |
| Letters to a Young Poet |      1934 |   1009 |
| Noncomformity            |      1996 |   1014 |
| Postcards                |      1992 |   1012 |
| The Shipping News        |      1993 |   1012 |
| Winesburg, Ohio          |      1919 |   NULL |
+--------------------------+-----------+--------+
9 rows in set (0.00 sec)
```

If you refer to the last row, you can see that a `NULL` value has been returned for the AuthID column. A `NULL` is returned because the book *Winesburg, Ohio* is listed in the Books table, but no author is listed in the Authors table. Whenever a value in the left table doesn't have an associated value in the right table, `NULL` is returned for the left table.

You can receive the same results as the preceding statement by replacing the `ON` clause with the `USING` clause, as shown in the following statement:

```
SELECT BookTitle, Copyright, AuthID
FROM Books LEFT JOIN AuthorBook
   USING (BookID)
ORDER BY BookTitle;
```

You can also use a left join to link more than two tables. For example, the following `SELECT` statement defines a left join on the Books, AuthorBook, and Authors tables:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b LEFT JOIN AuthorBook AS ab ON b.BookID=ab.BookID
   LEFT JOIN Authors AS a ON ab.AuthID=a.AuthID
ORDER BY BookTitle;
```

As you can see, the `FROM` clause includes an additional left join definition. Because of this, the result set can now include the actual author's name, rather than an ID, while preserving the left join, as shown in the following results:

```
+-------------------------+-----------+--------------------+
| BookTitle               | Copyright | Author             |
+-------------------------+-----------+--------------------+
| A Confederacy of Dunces |      1980 | John Kennedy Toole |
| Black Elk Speaks        |      1932 | Black Elk          |
| Black Elk Speaks        |      1932 | John G. Neihardt   |
| Hell's Angels           |      1966 | Hunter S. Thompson |
| Letters to a Young Poet |      1934 | Rainer Maria Rilke |
| Noncomformity           |      1996 | Nelson Algren      |
| Postcards               |      1992 | Annie Proulx       |
| The Shipping News       |      1993 | Annie Proulx       |
| Winesburg, Ohio         |      1919 |                    |
+-------------------------+-----------+--------------------+
9 rows in set (0.00 sec)
```

As you probably noticed, the Author column for the last row is now blank because you used the `CONCAT_WS()` function to concatenate the author names. Because the function cannot concatenate a value that doesn't exist, MySQL does not know to return `NULL`, so no value is inserted in the Author column. The results still tell you that *Winesburg, Ohio* is listed in the Books table, but no matching author is listed in the Authors table.

If you were to modify the preceding `SELECT` statement to remove the `CONCAT_WS()` function, your statement would now look similar to the following:

```
SELECT BookTitle, Copyright, AuthFN, AuthMN, AuthLN
FROM Books AS b LEFT JOIN AuthorBook AS ab ON b.BookID=ab.BookID
   LEFT JOIN Authors AS a ON ab.AuthID=a.AuthID
ORDER BY BookTitle;
```

Now MySQL interprets the columns as it would any column in the right table of a left join, as shown in the following result set:

```
+-------------------------+-----------+--------+---------+----------+
| BookTitle               | Copyright | AuthFN | AuthMN  | AuthLN   |
+-------------------------+-----------+--------+---------+----------+
| A Confederacy of Dunces |      1980 | John   | Kennedy | Toole    |
| Black Elk Speaks        |      1932 | Black  | NULL    | Elk      |
| Black Elk Speaks        |      1932 | John   | G.      | Neihardt |
| Hell's Angels           |      1966 | Hunter | S.      | Thompson |
| Letters to a Young Poet |      1934 | Rainer | Maria   | Rilke    |
| Noncomformity           |      1996 | Nelson | NULL    | Algren   |
| Postcards               |      1992 | Annie  | NULL    | Proulx   |
| The Shipping News       |      1993 | Annie  | NULL    | Proulx   |
| Winesburg, Ohio         |      1919 | NULL   | NULL    | NULL     |
+-------------------------+-----------+--------+---------+----------+
9 rows in set (0.00 sec)
```

As you can see, `NULL` is inserted in the AuthFN, AuthMN, and AuthLN columns of the last row of the result set.

Creating Right Joins

A right join is simply the counterpart to the left join and works the same as a left join, except that all values are retrieved from the table on the right. The following syntax shows how to define a right join in your `FROM` clause:

```
<table reference> RIGHT [OUTER] JOIN <table reference> [<join condition>]

<join condition>::=
ON <expression> [{<operator> <expression>}...]
| USING (<column> [{, <column>}...])
```

As you can see, the language is the same as with a left join, except for the keyword `RIGHT`. Now take a look at an example of a right join to demonstrate how to retrieve data from the joined tables. The following `SELECT` statement joins the Books table to the AuthorBook table, and the AuthorBook table to the Authors table:

```
SELECT BookTitle, Copyright, CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b RIGHT JOIN AuthorBook AS ab ON b.BookID=ab.BookID
   RIGHT JOIN Authors AS a ON ab.AuthID=a.AuthID
ORDER BY BookTitle;
```

As the statement shows, the first join condition is specified on the BookID column, and the second join condition is specified on the AuthID column. Although three tables are joined, the statement takes data only from the Books table and the Authors table, and the AuthorBook table is what links the other two tables together, with the Books table on the left side of the join condition and the Authors table on the right side. As a result, all authors are retrieved from the Authors table, but only books with matching BookID values (in Books and AuthorBook) are retrieved, as shown in the following result set:

```
+--------------------------+-----------+--------------------+
| BookTitle                | Copyright | Author             |
+--------------------------+-----------+--------------------+
| NULL                     |      NULL | Joyce Carol Oates  |
| NULL                     |      NULL | Alan Watts         |
| A Confederacy of Dunces  |      1980 | John Kennedy Toole |
| Black Elk Speaks         |      1932 | John G. Neihardt   |
| Black Elk Speaks         |      1932 | Black Elk          |
| Hell's Angels            |      1966 | Hunter S. Thompson |
| Letters to a Young Poet  |      1934 | Rainer Maria Rilke |
| Noncomformity            |      1996 | Nelson Algren      |
| Postcards                |      1992 | Annie Proulx       |
| The Shipping News        |      1993 | Annie Proulx       |
+--------------------------+-----------+--------------------+
10 rows in set (0.00 sec)
```

As you can see, the BookTitle and Copyright values in the first two rows are set to `NULL`. Because both columns are part of the Books table, the left table, only matched rows are retrieved. The Author column, which comes from the right table, includes all authors, even the two with no matching books in the Books table. If you had wanted, you could have reversed the order in which the tables are joined in the `FROM` clause and then changed the join definition to a left join, and you would have received the same results.

Now that you know how to create outer joins, in the following Try It Out you create a `SELECT` statement that uses a left join to retrieve data from the Employees table and the Orders table in the DVDRentals database.

Try It Out Using Outer Joins to Retrieve Data

Follow these steps to test out a left outer join:

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. In order to fully test a left join, you must first add data to the Employees table that provides you with results that demonstrate how a left join operates. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO Employees
VALUES (NULL, 'Rebecca', 'T.', 'Reynolds'),
(NULL, 'Charlie', 'Patrick', 'Waverly');
```

   You should receive a message indicating that the statement executed successfully, affecting two rows.

3. Now create a left outer join on the Employees and Order tables. Because the Employees table is named first in the join condition, all rows are retrieved from that table, whether or not any associated records exist in the Orders table. Execute the following SQL statement at the mysql command prompt:

```
SELECT CONCAT_WS(' ', EmpFN, EmpMN, EmpLN) AS Employee, OrderID
FROM Employees LEFT JOIN Orders USING (EmpID)
ORDER BY EmpLN;
```

   You should receive results similar to the following:

```
+-------------------------+---------+
| Employee                | OrderID |
+-------------------------+---------+
| George Brooks           |       4 |
| George Brooks           |      12 |
| Rita C. Carter          |       3 |
| Rita C. Carter          |       9 |
| John Laguci             |       8 |
| John Laguci             |      13 |
| Mary Marie Michaels     |       1 |
| Mary Marie Michaels     |       6 |
| Rebecca T. Reynolds     |    NULL |
| Robert Schroader        |       2 |
| Robert Schroader        |       7 |
| Robert Schroader        |      10 |
| John P. Smith           |       5 |
| John P. Smith           |      11 |
| Charlie Patrick Waverly |    NULL |
+-------------------------+---------+
15 rows in set (0.00 sec)
```

4. Now you should delete the rows that you added to the Employees table. Execute the following SQL statement at the mysql command prompt:

```
DELETE FROM Employees
WHERE EmpLN='Reynolds' OR EmpLN='Waverly';
```

You should receive a message indicating that the statement executed successfully, affecting two rows.

How It Works

In this exercise, you created a `SELECT` statement that defined a left join on the Employees and Orders tables, as shown in the following statement:

```
SELECT CONCAT_WS(' ', EmpFN, EmpMN, EmpLN) AS Employee, OrderID
FROM Employees LEFT JOIN Orders USING (EmpID)
ORDER BY EmpLN;
```

The join is defined in the `FROM` clause, which first lists the Employees table, the `LEFT JOIN` keywords, the Orders table, and the `USING` clause, which specifies that the join be defined on the EmpID column in both tables. Because the Employees table is listed first — on the left side of the join — all rows are retrieved from the Employees table, whether or not a matching order exists in the Orders table. On the other hand, the Orders table is on the right side of the join, so only rows with an EmpID value that equals the EmpID value in the Employees table are returned. As the result set shows, only two employees — Charlie Patrick Waverly and Rebecca T. Reynolds — are not linked with any orders in the Orders table.

Creating Natural Joins

Another type of join that you can specify in a `SELECT` statement is the natural join. This type of join can be specified as a full join, left join, or right join. The following syntax shows how to define a natural join in your `FROM` clause:

```
<table reference> NATURAL [{LEFT | RIGHT} [OUTER]] JOIN <table reference>
```

As you can see in the syntax, you do not define a join condition. A natural join automatically joins the columns in both tables that are common to each other. For example, if you were joining the Books table to the AuthorBook table, the natural join would automatically define that join on the BookID column, which is common to both tables.

To create a natural join, your `FROM` clause must include a table reference, the `NATURAL JOIN` keywords, and then a second table reference. If you include only these elements, you're creating a natural full join. If you also specify the `LEFT` or `LEFT OUTER` keywords, you're creating a natural left join. If you specify the `RIGHT` or `RIGHT OUTER` keywords, you're creating a natural right join.

First take a look at a natural full join to demonstrate how this works. In the following `SELECT` statement, the `FROM` clause defines a natural full join on the Books, AuthorBook, and Authors tables:

```
SELECT BookTitle, Copyright,
   CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b NATURAL JOIN AuthorBook AS ab
   NATURAL JOIN Authors AS a
WHERE Copyright<1980
ORDER BY BookTitle;
```

As you can see, this statement defines no join condition. There are no `ON` clauses or `USING` clauses, and the `WHERE` clause contains only an expression that limits rows to those whose Copyright value is less than 1980. There are no join conditions specified in the `WHERE` clause. When you execute this statement, you receive results similar to the following:

```
+-------------------------+-----------+--------------------+
| BookTitle               | Copyright | Author             |
+-------------------------+-----------+--------------------+
| Black Elk Speaks        |      1932 | Black Elk          |
| Black Elk Speaks        |      1932 | John G. Neihardt   |
| Hell's Angels           |      1966 | Hunter S. Thompson |
| Letters to a Young Poet |      1934 | Rainer Maria Rilke |
+-------------------------+-----------+--------------------+
4 rows in set (0.00 sec)
```

As you can see, the natural join automatically matches the BookID values in the Books and AuthorBook tables and matches the AuthID values in the Authors and AuthorBook tables. The natural join is similar to full joins in that it returns only matched rows, unlike outer joins that return unmatched values from either the left or right table.

If you want to create an outer join, rather than a full join, you simply add the `LEFT` or `RIGHT` keyword, as appropriate. For example, the following `SELECT` statement is similar to the preceding one, except for the addition of the `LEFT` keyword to each join definition:

```
SELECT BookTitle, Copyright,
   CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Books AS b NATURAL LEFT JOIN AuthorBook AS ab
   NATURAL LEFT JOIN Authors AS a
WHERE Copyright<1980
ORDER BY BookTitle;
```

As with the preceding statement, the BookID values and AuthID values are automatically matched. In addition, unmatched rows in the left table (the Books table) are also returned, as shown in the following result set:

```
+-------------------------+-----------+--------------------+
| BookTitle               | Copyright | Author             |
+-------------------------+-----------+--------------------+
| Black Elk Speaks        |      1932 | Black Elk          |
| Black Elk Speaks        |      1932 | John G. Neihardt   |
| Hell's Angels           |      1966 | Hunter S. Thompson |
| Letters to a Young Poet |      1934 | Rainer Maria Rilke |
| Winesburg, Ohio         |      1919 |                    |
+-------------------------+-----------+--------------------+
5 rows in set (0.00 sec)
```

As you can see, the book *Winesburg, Ohio* has no matching author in the Authors table, so a blank is displayed. As you recall, a blank is displayed, rather than `NULL`, because the `CONTACT_WS()` function concatenates the values in that column.

Joining Tables in an UPDATE Statement

In Chapter 6, you learned how to create an UPDATE statement that joins multiple tables in that statement. Basic joins were demonstrated in that chapter. MySQL allows you to use any join definition in your UPDATE statement. To reflect the ability to use different types of join definitions, the syntax for the UPDATE statement must be modified a bit from what you saw in Chapter 6. The updated syntax now shows how you can use different join definitions:

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

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

The syntax here shows only those components relevant to a multiple-table update. (For more information about the UPDATE statement, see Chapter 6.) The original syntax didn't use the <join definition> placeholder. Instead, you saw only the syntax for a basic join:

```
<table name> [{, <table name>}...]
```

By replacing this with <join definition>, you can see how to insert other types of joins in this part of the syntax. To demonstrate how this works, take a look at a couple of examples. First, assume that the following INSERT statements have been used to insert data in the Books table and the AuthorBook table:

```
INSERT INTO Books VALUES (21356, 'Tao: The Watercourse Way', 1975);
INSERT INTO AuthorBook VALUES (1013, 21356);
```

The author for this book (Alan Watts) is already listed in the Authors table, so that author is now linked to a book in the Books table. You can now perform an update based on that link. For example, the following UPDATE statement updates the last name of the author, based on the name of the book:

```
UPDATE Authors AS a, AuthorBook AS ab, Books AS b
SET AuthLN='Wats'
WHERE a.AuthID=ab.AuthID AND ab.BookID=b.BookID
   AND BookTitle='Tao: The Watercourse Way';
```

In this statement, the join is defined in the UPDATE clause. The statement uses a basic join, similar to the type of join used in the examples in Chapter 6. Notice that the join condition is specified in the WHERE clause. The AuthID values are linked, and the BookID values are linked. In addition, the WHERE clause includes an expression that specifies that the BookTitle value must be *Tao: The Watercourse Way*. As a result, all three conditions in the WHERE clause must be met in order for a row to be updated. In other words, for any updated row, the AuthID value in the Authors and AuthorBook tables must be equal, the BookID value in the Books and BookTitle table must be equal, and the BookTitle value must equal *Tao: The Watercourse Way*. When all three conditions are met, the AuthLN value is changed to Wats.

The author's name is being intentionally misspelled here, but it is corrected in the next example UPDATE *statement.*

You can modify the UPDATE statement so that a different join is defined in the UPDATE clause. For example, the following statement defines a CROSS join rather than a basic join:

```
UPDATE Authors CROSS JOIN AuthorBook USING (AuthID)
   CROSS JOIN Books USING (BookID)
SET AuthLN='Watts'
WHERE BookTitle='Tao: The Watercourse Way';
```

As you can see, two cross joins are now defined in the UPDATE clause. In addition, the USING clause specifies that the first join should be created on the AuthID column, and the second USING clause specifies that the second join should be created on the BookID column. This statement produces the same results as the previous UPDATE statement, except that the author's name is now correctly spelled.

> *MySQL product documentation recommends against performing multiple-table updates against* InnoDB *tables joined through foreign key constraints because you cannot always predict how the join optimizer might process tables. If an update is not performed in the correct order, the statement could fail. For* InnoDB *tables, it is generally recommended that you rely on the* ON UPDATE *options provided by* InnoDB *tables to modify the data.*

Joining Tables in a DELETE Statement

As you also saw in Chapter 6, you can define a join condition in a DELETE statement, and you can use any of the join definitions you saw for the SELECT statement. As a result, the syntax for the DELETE statement must be slightly modified to reflect its support for any join definition, as shown in the following syntax:

```
<delete statement>::=
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
{<single table delete> | <from join delete> | <using join delete>

<from join delete>::=
<table name>[.*] [{, <table name>[.*]}...]
FROM <join definition>
[WHERE <where definition>]

<using join delete>::=
FROM <table name>[.*] [{, <table name>[.*]}...]
USING <join definition>
[WHERE <where definition>]
```

Only the elements specific to joining tables in a DELETE statement are shown here. (For more information about the DELETE statement, see Chapter 6.) Also, in the original syntax, the <join definition> placeholder wasn't used. Instead, you saw the following syntax for the <from join delete> option:

```
FROM <table name> [{, <table name>}...]
```

For the <using join delete> option, the following syntax was used:

```
USING <table name> [{, <table name>}...]
```

In both cases, a basic join is reflected by the syntax. As the `<join definition>` placeholder demonstrates, you can use any join definition in these clauses. The following couple of examples help explain how the join definitions work. The first example uses a basic join to delete data from the AuthorBook and Authors tables:

```
DELETE ab
FROM AuthorBook AS ab, Authors AS a
WHERE ab.AuthID=a.AuthID AND AuthLN='Watts';
```

As you can see, the AuthorBook table is joined to the Authors table (in the `FROM` clause), and the join condition is defined in the `WHERE` clause (`ab.AuthID=a.AuthID`). The `WHERE` clause also identifies the name of the author, which is used to find the AuthID value for that author. As a result, the only rows deleted from the AuthorBook table are those whose AuthID values are the same in the AuthorBook and Authors tables whose AuthLN value is Watt.

You can also delete rows from multiple tables, as shown in the following example:

```
DELETE ab, b
FROM Authors AS a, AuthorBook AS ab, Books AS b
WHERE a.AuthID=ab.AuthID AND ab.BookID=b.BookID
   AND AuthLN='Watts';
```

In this case, a basic join is created on the Authors, AuthorBook, and Books table, and rows are deleted from the AuthorBook and Books tables. The join conditions are specified in the `WHERE` clause, along with a third condition that limits the rows deleted to those associated with the author whose last name is Watts. As a result, any rows that reference books written by Alan Watts are deleted from the AuthorBook and Books tables; the author is not deleted from the Authors table.

You can rewrite this statement to use an inner join, rather than a basic join, as shown in the following statement:

```
DELETE ab, b
FROM Authors AS a INNER JOIN AuthorBook AS ab ON a.AuthID=ab.AuthID
   INNER JOIN Books AS b ON ab.BookID=b.BookID
WHERE AuthLN='Watts';
```

Now the join conditions are specified in the `ON` clause of each inner join definition, but the `WHERE` clause still includes the expression that specifies that the AuthLN value must be Watts.

> *MySQL product documentation recommends against performing multiple-table deletions against* `InnoDB` *tables that are joined through foreign key constraints because you cannot always predict how the join optimizer might process tables. If an update is not performed in the correct order, the statement could fail. For* `InnoDB` *tables, it is generally recommended that you rely on the* `ON UPDATE` *options provided by* `InnoDB` *tables to modify the data.*

As you have seen, MySQL provides a number of different ways to join tables. Joins, however, are not the only method you can use to access multiple tables in a single statement. MySQL also supports the use of subqueries in your statements.

Creating Subqueries in Your SQL Statements

Another useful method that you can use to access multiple tables from within your SQL statement is to include a subquery in that statement. A *subquery*, also referred to as a *subselect*, is a SELECT statement that is embedded in another SQL statement. The results returned by the subquery — the inner statement — are then used by the outer statement in the same way a literal value would be used.

You can add subqueries to SELECT, UPDATE, and DELETE statements. In addition, you can embed subqueries in your statement to multiple levels. For example, one subquery can be embedded in another subquery, which is then embedded in still another subquery. In this section, you learn how you can add subqueries to your SELECT, UPDATE and DELETE statements.

> *Note that you can include* SELECT *statements in your INSERT and* CREATE TABLE *statements, but these* SELECT *statements are not considered subqueries. For a discussion of how to add* SELECT *statements to your* INSERT *and* CREATE TABLE *statements, see Chapter 11.*

Adding Subqueries to Your SELECT Statements

The most common way to include a subquery in a SELECT statement is to add it as an element of the WHERE clause to help restrict which rows the outer SELECT statement returns. When adding a subquery to a WHERE clause, you can use a comparison operator to introduce the subquery or you can use a subquery operator. In addition, you can use subqueries when using a GROUP BY clause in your SELECT statement. In this case, you can add the subquery to the HAVING clause of the statement. The following sections examine the different ways that you can use subqueries.

Working with Comparison Operators

One of the easiest ways to use a subquery in your SELECT statement is to include it as part of an expression in the WHERE clause of your statement. The subquery is included as one of the elements of expression and is introduced by a comparison operator. For example, the following SELECT statement uses a subquery to return an AuthID value:

```
SELECT CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Authors
WHERE AuthID=
   (
      SELECT ab.AuthID
      FROM AuthorBook AS ab, Books AS b
      WHERE ab.BookID=b.BookID AND BookTitle='Noncomformity'
   );
```

As you can see, the WHERE clause of the outer statement includes an expression that specifies that the AuthID value should equal the subquery. The subquery is the portion of the expression enclosed in parentheses. Subqueries must always be enclosed in parentheses. The subquery itself is a typical SELECT statement. Whenever a comparison operator precedes a subquery, the subquery must return a single value. That value is then compared to the first part of the expression.

> *Note that the example is based on the same three tables that you saw earlier in the chapter, as they were originally populated. The three tables — Books, AuthorBook, and Authors — are shown in Figure 10-1. You can also refer to the table definitions and* INSERT *statements that precede the figure to see how these tables are set up.*

Now take a look at the subquery itself to better understand how this works. Suppose you were to execute the subquery as an independent `SELECT` statement, as shown in the following statement:

```
SELECT ab.AuthID
FROM AuthorBook AS ab, Books AS b
WHERE ab.BookID=b.BookID AND BookTitle='Noncomformity';
```

The statement uses a join to retrieve the AuthID value from the AuthorBook table. If you were to execute this statement, you would receive the following results:

```
+--------+
| AuthID |
+--------+
|   1014 |
+--------+
1 row in set (0.00 sec)
```

Notice that the result set includes only one column and one value for that column. If your query were to return more that one value, you would receive an error when you tried to execute the outer `SELECT` statement. Because only one value is returned, the outer statement can now use that value. As a result, the `WHERE` clause in the outer statement now has a value to work with. The expression can now be interpreted as `AuthID=1014`. Once the subquery returns the value, the outer statement can be processed and can return the name of the author with an AuthID value of 1014, as shown in the following results.

```
+---------------+
| Author        |
+---------------+
| Nelson Algren |
+---------------+
1 row in set (0.01 sec)
```

An important point to make about subqueries is that they can often be rewritten as joins. For example, you can retrieve the same results as the preceding statement by using the following `SELECT` statement:

```
SELECT DISTINCT CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Authors AS a JOIN AuthorBook AS ab ON a.AuthID=ab.AuthID
   JOIN Books AS b ON ab.BookID=b.BookID
WHERE BookTitle='Noncomformity';
```

In this case, rather than using a subquery to retrieve the data, you use a join. In some cases, joins perform better than subqueries, so it's often more efficient to rewrite a subquery as a join when possible. If you're joining numerous tables and a subquery can bypass many of those joins, performance might be better with a subquery. When creating complex statements that retrieve large numbers of rows, it is sometimes best to create both types of statements, then compare the performance of each.

Another advantage of using a join in this case, rather than a subquery, is that the subquery fails if more than one author has written the specified book. For example, suppose you modify your subquery to specify a different book:

```
SELECT CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Authors
WHERE AuthID=
   (
```

```
        SELECT ab.AuthID
        FROM AuthorBook AS ab, Books AS b
        WHERE ab.BookID=b.BookID AND BookTitle='Black Elk Speaks'
    );
```

As it turns out, two authors wrote *Black Elk Speaks*, so the statement would return an error similar to the following:

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

Despite some of the limitations of using subqueries, they can often provide an efficient method for retrieving data, so take a look at some more examples of how they're used. In the last example, the equals (=) comparison operator is used in the `WHERE` clause expression of the outer statement to compare the results of the subquery. You can use other types of comparison operators. For example, the following statement uses a not equal (<>) comparison operator in the `WHERE` clause to introduce the subquery:

```
SELECT DISTINCT CONCAT_WS(' ', AuthFN, AuthMN, AuthLN) AS Author
FROM Authors
WHERE AuthID<>
   (
      SELECT ab.AuthID
      FROM AuthorBook AS ab, Books AS b
      WHERE ab.BookID=b.BookID AND BookTitle='Noncomformity'
   )
ORDER BY AuthLN;
```

Now rows returned by the outer `SELECT` statement *cannot* contain an AuthID value that is the same as the AuthID value assigned to author of *Noncomformity*. In other words, AuthID cannot equal 1014. As a result, the outer `SELECT` statement returns the following result set:

```
+--------------------+
| Author             |
+--------------------+
| Black Elk          |
| John G. Neihardt   |
| Joyce Carol Oates  |
| Annie Proulx       |
| Rainer Maria Rilke |
| Hunter S. Thompson |
| John Kennedy Toole |
| Alan Watts         |
+--------------------+
8 rows in set (0.00 sec)
```

Now look at another way that you can use a subquery. In the following example, the subquery uses an aggregate function to arrive at a value that the outer statement can use:

```
SELECT BookTitle, Copyright
FROM Books
WHERE Copyright<(SELECT MAX(Copyright)-50 FROM Books)
ORDER BY BookTitle;
```

In this statement, the subquery uses the `MAX()` aggregate function to determine the most recent year in the Copyright column and then subtracts 50 from that year. You can see what this value would be by executing the following subquery `SELECT` statement separately from the outer statement:

```
SELECT MAX(Copyright)-50 FROM Books;
```

The subquery returns the following result set:

```
+-------------------+
| MAX(Copyright)-50 |
+-------------------+
|              1946 |
+-------------------+
1 row in set (0.03 sec)
```

The subquery takes the highest value in the Copyright column (1996), subtracts 50, and returns a value of 1946. That value is then used in the `WHERE` clause expression of the outer statement. The expression can be interpreted as Copyright<1946. As a result, only rows with a copyright value that is less than 1946 are included in the result set, as shown in the following results:

```
+-------------------------+-----------+
| BookTitle               | Copyright |
+-------------------------+-----------+
| Black Elk Speaks        |      1932 |
| Letters to a Young Poet |      1934 |
| Winesburg, Ohio         |      1919 |
+-------------------------+-----------+
3 rows in set (0.00 sec)
```

An interesting aspect of the subquery used in the previous statement is that it retrieves a value from the same table as the outer statement. A subquery can provide a handy way to calculate a value that the outer statement can then use to specify which rows are returned.

Working with Subquery Operators

In the examples that you have looked at so far, each subquery had to return a single value in order to be used by the outer statement. MySQL provides a set of operators that allows you to work with subqueries that return multiple values. When using these operators, the subquery must still return a single column of values, but that column can contain more than one value.

The ANY and SOME Operators

The `ANY` and `SOME` operators, which are synonymous, allow you to create an expression that compares a column to any of the values returned by a subquery. For the expression to evaluate to true, any value returned by the subquery can be used. For example, the following statement returns book information about any books that have a copyright date greater than any books written by Proulx:

```
SELECT BookTitle, Copyright
FROM Books
WHERE Copyright > ANY
   (
      SELECT b.copyright
```

```
        FROM Books AS b JOIN AuthorBook AS ab USING (BookID)
           JOIN Authors AS a USING (AuthID)
        WHERE AuthLN='Proulx'
     )
ORDER BY BookTitle;
```

In this case, the subquery returns a list of Copyright values (1992 and 1993) for the books written by the author Proulx. Those values are used in the `WHERE` clause expression, which can be interpreted as `Copyright > ANY (1992, 1993)`. Because the statement uses the `ANY` operator, a value in the Copyright column must be greater than either 1992 or 1993, so the expression can now be interpreted as `Copyright>1992 OR Copyright>1993`. As a result, the outer query returns two rows, as shown in the following result set:

```
+-------------------+-----------+
| BookTitle         | Copyright |
+-------------------+-----------+
| Noncomformity     |      1996 |
| The Shipping News |      1993 |
+-------------------+-----------+
2 rows in set (0.00 sec)
```

As you might have realized, the second row returned, *The Shipping News,* is written by Annie Proulx. Because the book has a copyright value of 1993, which is greater than 1992, it is included in the result set, even though it is one of the values returned by the subquery.

The ALL Operator

The `ALL` operator is different from the `ANY` and `SOME` operators because it requires that all values returned by the subquery must cause the expression to evaluate to true before the outer statement can return a row. For example, modify the previous example to use the `ALL` operator rather than the `ANY` operator:

```
SELECT BookTitle, Copyright
FROM Books
WHERE Copyright > ALL
   (
      SELECT b.copyright
      FROM Books AS b JOIN AuthorBook AS ab USING (BookID)
         JOIN Authors AS a USING (AuthID)
      WHERE AuthLN='Proulx'
   )
ORDER BY BookTitle;
```

The subquery still returns the same values (1992 and 1993); however, the values in the Copyright column must now be greater than all the values returned by the subquery. In other words, the Copyright value must be greater than 1992 and greater than 1993. As a result, you can interpret the `WHERE` clause expression as `Copyright> 1992 AND Copyright>1993`. Now the result set includes only one row, as shown in the following:

```
+---------------+-----------+
| BookTitle     | Copyright |
+---------------+-----------+
| Noncomformity |      1996 |
+---------------+-----------+
1 row in set (0.00 sec)
```

The IN and NOT IN Operators

The `IN` and `NOT IN` operators provide the most flexibility when comparing values to the results returned by a subquery. To demonstrate how this works, replace the `ALL` operator and the comparison operator with the `IN` operator, as shown in the following statement:

```
SELECT BookTitle, Copyright
FROM Books
WHERE Copyright IN
   (
      SELECT b.copyright
      FROM Books AS b JOIN AuthorBook AS ab USING (BookID)
         JOIN Authors AS a USING (AuthID)
      WHERE AuthLN='Proulx'
   )
ORDER BY BookTitle;
```

Now the copyright value must equal either 1992 or 1993 because these are the two values returned by the subquery. As a result, the outer statement returns the following rows:

```
+-------------------+-----------+
| BookTitle         | Copyright |
+-------------------+-----------+
| Postcards         |      1992 |
| The Shipping News |      1993 |
+-------------------+-----------+
2 rows in set (0.00 sec)
```

You would return the same results if you were to use the equals comparison operator along with the `ANY` operator, as in `Copyright = ANY`, but the `IN` operator provides a simpler construction.

Now revise the `SELECT` statement to use the `NOT IN` operator:

```
SELECT BookTitle, Copyright
FROM Books
WHERE Copyright NOT IN
   (
      SELECT b.copyright
      FROM Books AS b JOIN AuthorBook AS ab USING (BookID)
         JOIN Authors AS a USING (AuthID)
      WHERE AuthLN='Proulx'
   )
ORDER BY BookTitle;
```

The results are now the opposite of those generated by the previous statement. The outer statement returns only those rows whose Copyright value is not equal to 1992 or 1993, as shown in the following result set:

```
+-------------------------+-----------+
| BookTitle               | Copyright |
+-------------------------+-----------+
| A Confederacy of Dunces |      1980 |
| Black Elk Speaks        |      1932 |
| Hell's Angels           |      1966 |
```

```
| Letters to a Young Poet |      1934 |
| Noncomformity           |      1996 |
| Winesburg, Ohio         |      1919 |
+-------------------------+-----------+
6 rows in set (0.00 sec)
```

Although the `IN` and `NOT IN` operators are very useful for comparing values returned by a subquery, you might find that you simply need to determine whether a subquery returns a value. The actual value is not important. To allow you to test for the existence of a value, MySQL provides the `EXISTS` and `NOT EXISTS` operators.

The EXISTS and NOT EXISTS Operators

The `EXISTS` and `NOT EXISTS` operators are very different from the previous subquery operators that you have seen. The `EXISTS` and `NOT EXISTS` operators are used only to test whether a subquery does or does not produce any results. If you use the `EXISTS` operator, and the subquery returns results, then the condition evaluates to true; otherwise, it is evaluates to false. If you use the `NOT EXISTS` operator, and the subquery returns results, the condition evaluates to false; otherwise, it evaluates to true.

The `EXISTS` and `NOT EXISTS` operators are useful primarily when trying to correlate conditions in the subquery with the outer statement. This is best explained through an example. Suppose that you want to retrieve information from the Books table about only those books associated with authors listed in the Authors table. Because the AuthorBook table already tells you whether a book is associated with an author, you can use that table to help you determine which book titles to return, as shown in the following `SELECT` statement:

```
SELECT BookID, BookTitle
FROM Books AS b
WHERE EXISTS
   (
      SELECT BookID
      FROM AuthorBook AS ab
      WHERE b.BookID=ab.BookID
   )
ORDER BY BookTitle;
```

The first thing to look at is the `WHERE` clause of the subquery. This is an unusual construction because it shows a join condition even though the subquery doesn't define a join. This is how the subquery is correlated with the outer table. The BookID value of the AuthorBook table, which is specified in the subquery, must equal the BookID value in the Books table, which is specified in the outer statement. As a result, the subquery returns a row only if that row contains a BookID value that correlates to a BookID value of the outer table. If a row is returned, the condition specified in the `WHERE` clause evaluates to true. If a row is not returned, the condition evaluates to false. As a result, the outer statement returns only those rows whose BookID values exist in the AuthorBook table, as shown in the following result set:

```
+--------+-------------------------+
| BookID | BookTitle               |
+--------+-------------------------+
|  17695 | A Confederacy of Dunces |
|  15729 | Black Elk Speaks        |
|  14356 | Hell's Angels           |
|  12786 | Letters to a Young Poet |
```

```
|  16284 | Noncomformity           |
|  19264 | Postcards               |
|  19354 | The Shipping News       |
+--------+-------------------------+
7 rows in set (0.00 sec)
```

For each book returned, a BookID exists in the AuthorBook table. If a book is listed in the Books table and the BookID value for that book is not included in the AuthorBook table, the book is not displayed.

You can use the `NOT EXISTS` operator in the same way as the `EXISTS` operator, only the opposite results are returned. The following `SELECT` statement is identical to the preceding one except that the `WHERE` clause of the outer statement now includes the `NOT` keyword:

```
SELECT BookID, BookTitle
FROM Books AS b
WHERE NOT EXISTS
   (
      SELECT BookID
      FROM AuthorBook AS ab
      WHERE b.BookID=ab.BookID
   )
ORDER BY BookTitle;
```

As you would expect, the results returned by this statement include any books whose BookID value is not listed in the AuthorBook table. In other words, any book not associated with an author is returned, as shown in the following result set:

```
+--------+-----------------+
| BookID | BookTitle       |
+--------+-----------------+
|  13331 | Winesburg, Ohio |
+--------+-----------------+
1 row in set (0.00 sec)
```

Working with Grouped Data

The subqueries that you have seen so far have been used in the `WHERE` clauses of various `SELECT` statements. You can also use subqueries in a `HAVING` clause when you group data together. For the most part, adding subqueries to your `HAVING` clause is similar to using them in the `WHERE` clause, except that you're working with grouped data. To demonstrate how this works, take a look at a couple of examples. The examples use the same three tables (Books, AuthorBook, and Authors) that have been used in the previous examples. Yu also need an additional table, which is shown in the following table definition:

```
CREATE TABLE BookOrders
(
   OrderID SMALLINT NOT NULL,
   BookID SMALLINT NOT NULL,
   Quantity SMALLINT NOT NULL,
   PRIMARY KEY (OrderID, BookID),
   FOREIGN KEY (BookID) REFERENCES Books (BookID)
)
ENGINE=INNODB;
```

For these exercises, you can assume that the following `INSERT` statement populated the table:

```
INSERT INTO BookOrders
VALUES (101, 13331, 1), (101, 12786, 1), (101, 16284, 2), (102, 19354, 1),
(102, 15729, 3), (103, 12786, 2), (103, 19264, 1), (103, 13331, 1),
(103, 14356, 2), (104, 19354, 1), (105, 15729, 1), (105, 14356, 2),
(106, 16284, 2), (106, 13331, 1), (107, 12786, 3), (108, 19354, 1),
(108, 16284, 4), (109, 15729, 1), (110, 13331, 2), (110, 12786, 2),
(110, 14356, 2), (111, 14356, 2);
```

Figure 10-3 shows you how the four tables are related.

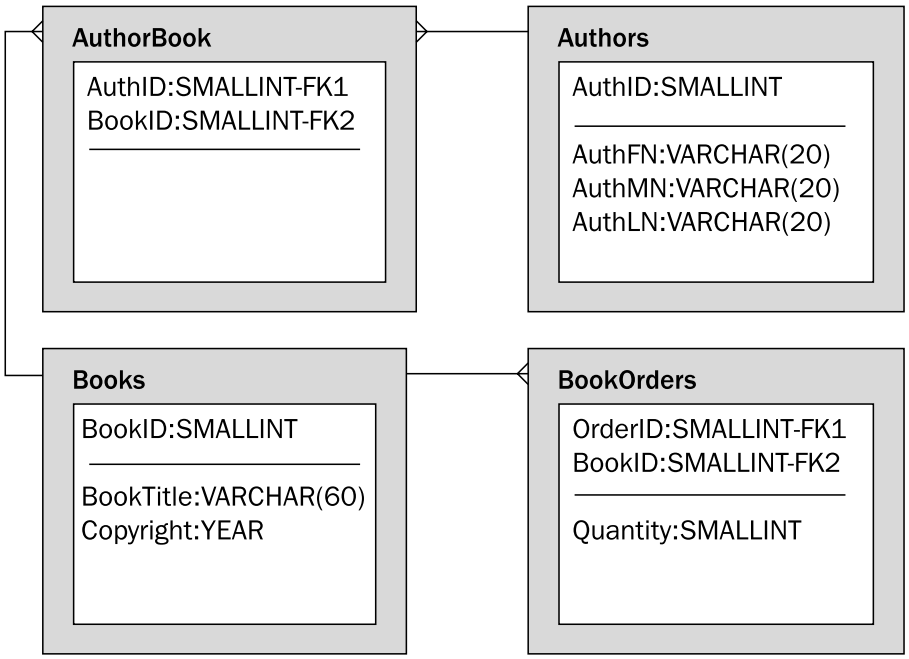

Figure 10-3

Now take a look at an example. Suppose that you want to retrieve the total number of books sold for each order listed in the BookOrders table. To do so, you would group together the data based on the OrderID column and then use the `SUM()` function to retrieve the number of books for each group. Now suppose that you want the result set to include only those groups whose total number of books sold is greater than the average number of books sold for each title in each order. You can use a subquery to find that average, then use the average in the `HAVING` clause of the outer statement, as shown in the following statement:

```
SELECT OrderID, SUM(Quantity) AS Total
FROM BookOrders
GROUP BY OrderID
HAVING Total>(SELECT AVG(Quantity) FROM BookOrders);
```

As you can see, the statement includes a `GROUP BY` clause that specifies which column should be grouped together and a `HAVING` clause that specifies which rows of grouped data to return. Notice that

the expression in the having clause is made up of the Total column (defined in the SELECT clause), the greater than (>) comparison operator, and a subquery. The subquery returns that average value of the Quantity column of the BookOrders table, which is a value of 1.7273. In other words, the average number of books sold for each title in an order is 1.7273.

The outer statement then uses this value in the HAVING clause. As a result, the HAVING expression can be interpreted as Total>1.7273. This means that every order that has sold more that 1.7273 books is included in the result set, as shown in the following:

```
+---------+-------+
| OrderID | Total |
+---------+-------+
|     101 |     4 |
|     102 |     4 |
|     103 |     6 |
|     105 |     3 |
|     106 |     3 |
|     107 |     3 |
|     108 |     5 |
|     110 |     6 |
|     111 |     2 |
+---------+-------+
9 rows in set (0.00 sec)
```

As you can see, no orders are listed that contain fewer than two books. Now suppose that you want to determine how many books have been ordered for authors who have more than one book listed in the Books table. Because you don't need to know the name of the author, you can go directly to the AuthorBook table to determine which authors are associated with more than one book. From there, you can determine the BookID value for those books, then use that value to determine which books in the BookOrders table to include in the result set. The following SELECT statement shows you how this can be accomplished:

```
SELECT BookID, SUM(Quantity) AS Total
FROM BookOrders
GROUP BY BookID
HAVING BookID IN
   (SELECT BookID FROM AuthorBook WHERE AuthID IN
      (
         SELECT AuthID FROM AuthorBook
         GROUP BY AuthID
         HAVING COUNT(*)>1
      )
   );
```

The first thing that you might notice is that a subquery is embedded in another subquery. So take a look at the most deeply embedded subquery first:

```
SELECT AuthID FROM AuthorBook
GROUP BY AuthID
HAVING COUNT(*)>1;
```

If you were to execute this statement separately from the other parts of the statement, you would receive the following results:

```
+--------+
| AuthID |
+--------+
|   1012 |
+--------+
1 row in set (0.00 sec)
```

The subquery is requesting the AuthID value for any group of AuthID values that have a value greater than 1. In this case, only author 1012 is associated with more than one BookID value in the AuthorBook table. Now that you have retrieved the AuthID value that you need, you can use it in the second subquery, as shown in the following:

```
SELECT BookID FROM AuthorBook WHERE AuthID IN
   (
      SELECT AuthID FROM AuthorBook
      GROUP BY AuthID
      HAVING COUNT(*)>1
   );
```

Notice that the `WHERE` clause in the outer subquery includes an expression specifying that the AuthID value must be one of the values returned by the inner subquery. In this case, only one value (1012) is returned, so any rows returned by the outer subquery must have an AuthID value of 1012. If you were to execute the outer and inner subqueries together, you would receive the following results:

```
+--------+
| BookID |
+--------+
|  19264 |
|  19354 |
+--------+
2 rows in set (0.00 sec)
```

Notice that two BookID values are returned. The values represent the two books that were written by the author with the AuthID value of 1012. The outer `SELECT` statement can then use these values in the `HAVING` clause expression to determine which rows are returned. Because of the `HAVING` clause, only groups that have a BookID value of 19264 or 19354 are returned, along with the total number of books sold, as shown in the following result set:

```
+--------+-------+
| BookID | Total |
+--------+-------+
|  19264 |     1 |
|  19354 |     3 |
+--------+-------+
2 rows in set (0.01 sec)
```

Had more than one author written more than one book, those rows would appear here as well. By using subqueries in the `HAVING` clause, you can be very specific about exactly which groups to include in the result set.

In the following Try It Out exercise, you create several SELECT statements that include subqueries that retrieve data that the outer SELECT statements can then use.

Try It Out Including Subqueries in Your SELECT Statements

The following steps explain how to include subqueries in SELECT statements:

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. The first SELCT statement that you create retrieves data from the DVDs table. The SELECT statement uses a subquery to return a value from the Status value, which the outer statement then uses. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName
FROM DVDs
WHERE StatID<>
   (SELECT StatID FROM Status WHERE StatDescrip='Available')
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+--------------------+
| DVDName            |
+--------------------+
| A Room with a View |
| Out of Africa      |
| White Christmas    |
+--------------------+
3 rows in set (0.00 sec)
```

3. The next SELECT statement that you create retrieves data from the DVDs and MovieTypes tables. The statement also contains a subquery in the WHERE clause of the outer statement that retrieves a value from the Status table, which the outer statement then uses. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip As MovieType
FROM DVDs AS d, MovieTypes AS mt
WHERE d.MTypeID=mt.MTypeID AND StatID=
   (SELECT StatID FROM Status WHERE StatDescrip='Available')
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+------------------------------+-----------+
| DVDName                      | MovieType |
+------------------------------+-----------+
| Amadeus                      | Drama     |
| Mash                         | Comedy    |
| The Maltese Falcon           | Drama     |
| The Rocky Horror Picture Show | Comedy    |
| What's Up, Doc?              | Comedy    |
+------------------------------+-----------+
5 rows in set (0.00 sec)
```

4. The next `SELECT` statement that you create is similar to the last, except that the `WHERE` clause of the outer statement uses the `IN` operator to compare values to those returned by the subquery. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip As MovieType
FROM DVDs AS d, MovieTypes AS mt
WHERE d.MTypeID=mt.MTypeID AND StatID IN
   (SELECT StatID FROM Status WHERE StatDescrip<>'Available')
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+--------------------+-----------+
| DVDName            | MovieType |
+--------------------+-----------+
| A Room with a View | Drama     |
| Out of Africa      | Drama     |
| White Christmas    | Musical   |
+--------------------+-----------+
3 rows in set (0.01 sec)
```

5. The last `SELECT` statement that you create uses a `GROUP BY` clause to group data together. In addition, the statement includes a `HAVING` clause that contains a subquery that returns data from the MovieTypes table. Execute the following SQL statement at the mysql command prompt:

```
SELECT MTypeID, RatingID, COUNT(*) AS TotalDVDs
FROM DVDs
GROUP BY MTypeID, RatingID
HAVING MTypeID IN
   (SELECT MTypeID FROM MovieTypes WHERE MTypeDescrip<>'Documentary');
```

You should receive results similar to the following:

```
+---------+----------+-----------+
| MTypeID | RatingID | TotalDVDs |
+---------+----------+-----------+
| mt11    | NR       |         2 |
| mt11    | PG       |         2 |
| mt12    | G        |         1 |
| mt12    | NR       |         1 |
| mt12    | R        |         1 |
| mt16    | NR       |         1 |
+---------+----------+-----------+
6 rows in set (0.00 sec)
```

How It Works

The first `SELECT` statement that you created in this exercise includes a subquery in the `WHERE` clause:

```
SELECT DVDName
FROM DVDs
WHERE StatID<>
   (SELECT StatID FROM Status WHERE StatDescrip='Available')
ORDER BY DVDName;
```

The WHERE clause in the outer SELECT statement includes an expression that is made up of the StatID column, the not equal (<>) comparison operator, and the following subquery:

```
SELECT StatID FROM Status WHERE StatDescrip='Available';
```

If you were to execute the subquery independently of the rest of the statement, you would receive the following results:

```
+--------+
| StatID |
+--------+
| s2     |
+--------+
1 row in set (0.00 sec)
```

The outer SELECT statement uses those results to complete the expression in the WHERE clause. As a result, only rows that have a StatID value of s2 are included in the result set.

The next SELECT statement that you created is similar to the last one except that it now includes a join definition in the outer statement:

```
SELECT DVDName, MTypeDescrip As MovieType
FROM DVDs AS d, MovieTypes AS mt
WHERE d.MTypeID=mt.MTypeID AND StatID=
   (SELECT StatID FROM Status WHERE StatDescrip='Available')
ORDER BY DVDName;
```

The FROM clause in the outer statement joins together the DVDs table and the MovieTypes table. In addition, MTypeDescrip values are now included in the result set. Like the previous SELECT statement, only rows with a StatID value of s2 are included in the result set because that is the value returned by the subquery.

The next SELECT statement that you created is a little different from the last one in that it uses an IN operator in the outer WHERE clause to introduce the subquery:

```
SELECT DVDName, MTypeDescrip As MovieType
FROM DVDs AS d, MovieTypes AS mt
WHERE d.MTypeID=mt.MTypeID AND StatID IN
   (SELECT StatID FROM Status WHERE StatDescrip<>'Available')
ORDER BY DVDName;
```

The IN operator is used because the subquery has been modified and now returns multiple values. The subquery returns the StatID value of all rows in the Status table whose StatDescrip value does not equal Available, as shown in the following result set:

```
+--------+
| StatID |
+--------+
| s1     |
| s3     |
| s4     |
+--------+
3 rows in set (0.00 sec)
```

These values are then compared to the StatID value in the DVDs table. Any rows that have a StatID value that matches the values returned by the subquery are included in the result set.

The next SELECT statement that you created groups together data in the DVDs table according to the MTypeID and RatingID columns:

```
SELECT MTypeID, RatingID, COUNT(*) AS TotalDVDs
FROM DVDs
GROUP BY MTypeID, RatingID
HAVING MTypeID IN
   (SELECT MTypeID FROM MovieTypes WHERE MTypeDescrip<>'Documentary');
```

The statement includes a HAVING clause that contains a subquery. The subquery returns the MTypeID value for all rows in the MovieTypes table that do not have a MTypeDescrip value of Documentary, as shown in the following query results:

```
+---------+
| MTypeID |
+---------+
| mt10    |
| mt11    |
| mt12    |
| mt13    |
| mt14    |
| mt16    |
+---------+
6 rows in set (0.00 sec)
```

The HAVING clause then uses the subquery results to limit the groups returned in the result set to those that have an MTypeID value for any movie type except Documentary. Because you used the IN operator in the HAVING clause, the result set includes any MTypeID value that equals one of the values returned by the subquery.

Adding Subqueries to Your UPDATE Statements

Up to this point, you have used subqueries in SELECT statements. You can also include subqueries in your UPDATE statements. You can use subqueries in the SET clause to help define the new values to be inserted in a table, or you can use them in the WHERE clause to help define which rows should be updated. Take a look at an example to illustrate how to use a subquery in an UPDATE statement. For this example, assume that the following INSERT statements have been used to add data to the Books and AuthorBook table:

```
INSERT INTO Books VALUES (21356, 'Tao: The Watercourse Way', 1975);
INSERT INTO AuthorBook VALUES (1013, 21356);
```

Now take a look at an UPDATE statement that contains a subquery in its WHERE clause:

```
UPDATE Books
SET BookTitle='The Way of Zen', Copyright=1957
WHERE BookID=
   (
```

```
        SELECT ab.BookID
        FROM Authors AS a, AuthorBook AS ab
        WHERE a.AuthID=ab.AuthID AND a.AuthLN='Watts'
    );
```

The statement updates a row in the Books table. The subquery determines which row should be updated. In this case, the subquery returns a BookID value based on an author with a last name of Watts. The subquery defines a join condition that joins the Authors and AuthorBook tables and then retrieves the BookID value for the book written by Watts. The `WHERE` expression then uses that value in the outer statement to limit the updated rows. Only the row that has a BookID value that matches the value returned by the subquery is updated. As a result, the row in the Books table with a BookID value of 21356 and a BookTitle value of *Tao: The Watercourse Way* has been changed so that the BookTitle value is now *The Way of Zen*.

Also, you should keep in mind that the subquery must return only one value when it is used in a comparison as it is used here. If you had specified Proulx as the author (in the `WHERE` clause of the subquery), two BookID values would have been returned, which would have made the outer `UPDATE` statement fail.

Adding Subqueries to Your DELETE Statements

You can also add subqueries to the `WHERE` clause of a `DELETE` statement, just as you can with `UPDATE` and `SELECT` statements. The subquery helps to determine which rows should be deleted from a table. For example, the subquery in the following `DELETE` statement returns the AuthID value for the author with last name of Watts:

```
DELETE ab, b
FROM AuthorBook AS ab, Books AS b
WHERE ab.BookID=b.BookID
   AND ab.AuthID=(SELECT AuthID FROM Authors WHERE AuthLN='Watts');
```

One of the expressions in the `WHERE` clause of the outer statement uses the value returned by the subquery to specify which rows are deleted. Because the `DELETE` statement joins together the AuthorBook table and the Books table, only those rows whose BookID values are equal and whose AuthID value equals the value returned by the subquery are deleted from the tables. In other words, those rows in the AuthorBook table and the Books table that contains a BookID value associated with Alan Watts are deleted.

As this section illustrates, you can use subqueries in `UPDATE` and `DELETE` statements to determine which values should be modified in a table. In the next Try It Out exercise, you create both an `UPDATE` statement and a `DELETE` statement that use subqueries.

Try It Out — Including Subqueries in Your UPDATE and DELETE Statements

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. In order to demonstrate how to use subqueries in your `UPDATE` and `DELETE` statements, you should first add a row to the Employees table. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO Employees
VALUES (NULL, 'Rebecca', 'T.', 'Reynolds');
```

   You should receive a message indicating that the statement executed successfully, affecting one row.

3. When you inserted the row in the Employees table, a value was automatically inserted in the primary key column. Now use the `LAST_INSERT_ID()` function to retrieve the primary key value. Execute the following SQL statement at the mysql command prompt:

```
SELECT LAST_INSERT_ID();
```

   You should receive results similar to the following:

```
+------------------+
| last_insert_id() |
+------------------+
|                7 |
+------------------+
1 row in set (0.01 sec)
```

   Note that you will not necessarily receive a value of 7. Depending on whether you inserted any other rows in the table (whether or not they still exist) or whether you re-created the database or the table, the value might be anywhere from 7 on up. Whatever the actual value is, you should remember that value for use in later statements. For the purposes of this exercise, the value 7 is assumed.

4. Now you need to insert a row in the Orders table. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO Orders (CustID, EmpID) VALUES (6, 5);
```

   You should receive a message indicating that the statement executed successfully, affecting one row.

5. Again, you need to retrieve the last value inserted in the primary key column. Execute the following SQL statement at the mysql command prompt:

```
SELECT LAST_INSERT_ID();
```

   You should receive results similar to the following:

```
+------------------+
| last_insert_id() |
+------------------+
|               14 |
+------------------+
1 row in set (0.01 sec)
```

   Once again, the value returned might vary, depending on previous modifications to the table. The value that you receive will be anything from 14 on up. For the purposes of this exercise, the value 14 is assumed.

6. Now you create an UPDATE statement that updates the Orders table. The statement uses a subquery in the SET clause to retrieve the EmpID value from the Employees table. Execute the following SQL statement at the mysql command prompt:

```
UPDATE Orders
SET EmpID=(SELECT EmpID FROM Employees WHERE EmpLN='Reynolds')
WHERE OrderID=14;
```

   You should receive a message indicating that the statement executed successfully, affecting one row.

7. Execute the following SQL statement at the mysql command prompt:

```
DELETE FROM Orders
WHERE EmpID=(SELECT EmpID FROM Employees WHERE EmpLN='Reynolds');
```

   You should receive a message indicating that the statement executed successfully, affecting one row.

8. Finally, delete the row that you added to the Employees table. Execute the following SQL statement at the mysql command prompt:

```
DELETE FROM Employees
WHERE EmpLN='Reynolds';
```

   You should receive a message indicating that the statement executed successfully, affecting one row.

How It Works

In this exercise, you created an UPDATE statement that updated a row in the Orders table:

```
UPDATE Orders
SET EmpID=(SELECT EmpID FROM Employees WHERE EmpLN='Reynolds')
WHERE OrderID=14;
```

The row updated had an OrderID value of 14. For this row, the EmpID value was changed to the value returned by the subquery. The subquery returns the EmpID value associated with the employee whose last name is Reynolds. Because the value returned is 7, the EmpID value in the Orders table is set to 7.

Next, you created a DELETE statement that deleted the row that you just updated:

```
DELETE FROM Orders
WHERE EmpID=(SELECT EmpID FROM Employees WHERE EmpLN='Reynolds');
```

The DELETE statement includes a subquery in the WHERE clause that retrieves the employee ID for Reynolds. Because the subquery returns a value of 7, any rows in the Orders table with an EmpID value of 7 are deleted from the table.

Creating Unions That Join SELECT Statements

Times might arise when you want to combine the results of multiple SELECT statements in one result set. An easy way to do this is to use the UNION operator to connect the statements, as shown in the following syntax:

```
<select statement> UNION <select statement>
```

To join two statements in this way, the statements must return the same number of values, and the data types for the returned values must correspond to each other. For example, suppose that you are joining two SELECT statements. The first statement returns two columns: ColA and ColB. ColA is configured with an INT data type, and ColB is configured with a CHAR data type. As a result, the second query must also return two columns, and those columns must be configured with an INT data type and CHAR data type, respectively.

Now take a look at an example to demonstrate how this works. Suppose that a second table is added to the set of example tables that have been used in this chapter. The table is named Authors2 and is defined by the following CREATE TABLE statement:

```
CREATE TABLE Authors2
(
   AuthID SMALLINT NOT NULL PRIMARY KEY,
   AuthFN VARCHAR(20),
   AuthMN VARCHAR(20),
   AuthLN VARCHAR(20)
);
```

Now assume that the following INSERT statement populated the Authors2 table:

```
INSERT INTO Authors2
VALUES (1006, 'Mark', NULL, 'Twain'),
(2205, 'E.', 'M.', 'Forster'),
(2206, 'Gabriel', 'Garcia', 'Marquez'),
(2207, 'Raymond', NULL, 'Carver'),
(2208, 'Mary', NULL, 'Shelley'),
(2209, 'Albert', NULL, 'Camus');
```

Once you have similar tables from which to retrieve data, you can use the UNION operator to join SELECT statements, as shown in the following statement:

```
SELECT AuthFN, AuthMN, AuthLN FROM Authors
UNION
SELECT AuthFN, AuthMN, AuthLN FROM Authors2;
```

As you can see, using a UNION operator is very simple, as long as the number of columns and their data types match. When you execute the statement, the following results are returned:

```
+---------+---------+----------+
| AuthFN  | AuthMN  | AuthLN   |
+---------+---------+----------+
| Hunter  | S.      | Thompson |
| Joyce   | Carol   | Oates    |
| Black   | NULL    | Elk      |
| Rainer  | Maria   | Rilke    |
| John    | Kennedy | Toole    |
| John    | G.      | Neihardt |
| Annie   | NULL    | Proulx   |
| Alan    | NULL    | Watts    |
| Nelson  | NULL    | Algren   |
| Mark    | NULL    | Twain    |
| E.      | M.      | Forster  |
| Gabriel | Garcia  | Marquez  |
| Raymond | NULL    | Carver   |
| Mary    | NULL    | Shelley  |
| Albert  | NULL    | Camus    |
+---------+---------+----------+
15 rows in set (0.00 sec)
```

As you can see, the data from Authors and Authors 2 have been added together in one result set. You might want to sort the result set in a specific order. To do this, you can use the `ORDER BY` clause, as shown in the following example:

```
(SELECT AuthFN, AuthMN, AuthLN FROM Authors)
UNION
(SELECT AuthFN, AuthMN, AuthLN FROM Authors2)
ORDER BY AuthLN;
```

To use the `ORDER BY` clause, you must enclose each `SELECT` statement in a set of parentheses and then add the `ORDER BY` clause at the end of the statement. Now when you execute the statement, you should receive results similar to the following:

```
+---------+---------+----------+
| AuthFN  | AuthMN  | AuthLN   |
+---------+---------+----------+
| Nelson  | NULL    | Algren   |
| Albert  | NULL    | Camus    |
| Raymond | NULL    | Carver   |
| Black   | NULL    | Elk      |
| E.      | M.      | Forster  |
| Gabriel | Garcia  | Marquez  |
| John    | G.      | Neihardt |
| Joyce   | Carol   | Oates    |
| Annie   | NULL    | Proulx   |
| Rainer  | Maria   | Rilke    |
| Mary    | NULL    | Shelley  |
| Hunter  | S.      | Thompson |
| John    | Kennedy | Toole    |
| Mark    | NULL    | Twain    |
| Alan    | NULL    | Watts    |
+---------+---------+----------+
15 rows in set (0.00 sec)
```

In this section, you learned how to create a union to join two SELECT statements together to produce one set of query results. In the Try It Out exercise that follows, you create your own union statement.

Try It Out Using Unions to Join SELECT Statements

The following steps explain how to use a union to join two SELECT statements:

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. Before you can create a union, you must add one more table to the DVDRentals database that is similar to the original Employees table. Execute the following SQL statement at the mysql command prompt:

```
CREATE TABLE Employees2
   (
   EmpID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   EmpFN VARCHAR(20) NOT NULL,
   EmpLN VARCHAR(20) NOT NULL
   )
ENGINE=INNODB;
```

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

3. Now add two rows to the new table. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO Employees2
VALUES (NULL, 'Rebecca', 'Reynolds'),
(NULL, 'Charlie', 'Waverly');
```

You should receive a message indicating that the statement executed successfully, affecting two rows.

4. You can now use a union to join the Employees and Employees2 tables. Execute the following SQL statement at the mysql command prompt:

```
(SELECT EmpLN, EmpFN FROM Employees)
UNION
(SELECT EmpLN, EmpFN FROM Employees2)
ORDER BY EmpLN;
```

You should receive results similar to the following:

```
+-----------+---------+
| EmpLN     | EmpFN   |
+-----------+---------+
| Brooks    | George  |
| Carter    | Rita    |
| Laguci    | John    |
| Michaels  | Mary    |
| Reynolds  | Rebecca |
```

```
| Schroader | Robert  |
| Smith     | John    |
| Waverly   | Charlie |
+-----------+---------+
8 rows in set (0.00 sec)
```

5. Finally, remove the Employees2 table from the DVDRentals database. Execute the following SQL statement at the mysql command prompt:

```
DROP TABLE Employees2;
```

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

How It Works

In this exercise, you used a `UNION` operator to join together two `SELECT` statements:

```
(SELECT EmpLN, EmpFN FROM Employees)
UNION
(SELECT EmpLN, EmpFN FROM Employees2)
ORDER BY EmpLN;
```

The first `SELECT` statement retrieves the EmpLN and EmpFN values from the Employees table. The second `SELECT` statement retrieves the EmpLN and EmpFN values from the Employees2 table. You might have noticed that the Employees2 table contains a different number of columns than the Employees table. This does not affect the union query as long as each `SELECT` statement retrieves the same number of columns and those columns are configured with the same data type.

The statement also uses an `ORDER BY` clause to sort the query results according to the EmpLN values. As a result, each `SELECT` statement is enclosed in its own set of parentheses.

Summary

As you learned in this chapter, you can create SQL statements that allow you to access multiple tables in a single statement. This is useful when you need to access related data stored in different tables. For example, you can use a `SELECT` statement to join together multiple tables and then extract the data you need from any of those tables. The data you retrieve depends on how that data is related and on how you define the join in the `SELECT` statement. MySQL provides several methods for accessing multiple tables in your SQL statements. These include joins, subqueries, and unions. In this chapter, you learned how to use all three methods to access data in multiple tables. Specifically, you learned how to do the following:

- Create basic, inner, and straight joins that retrieve matched data from two or more tables
- Create left and right joins that retrieve matched data from two or more tables, plus additional data from one of the joined tables
- Create natural, full, and outer joins that automatically match data from one or more tables without having to specify a join condition
- Add subqueries to your `SELECT`, `UPDATE`, and `DELETE` statements

- Use subqueries in `SELECT` statements that group and summarize data
- Use the `UNION` operator to join `SELECT` statements together in order to return one result set

As the chapter demonstrated, joins, subqueries, and unions provide powerful tools for accessing related data in multiple tables. As you build applications that retrieve data from a MySQL database, you will use these methods often to access data. This is especially true of `SELECT` statements that define joins in those statements. In fact, for many applications, this is the most commonly used type of SQL statement. As a result, it is well worth your time to experiment with the `SELECT` statements that include different types of joins. The more you use these statements, the greater your understanding of their flexibility and precision. Once you're comfortable with joining tables, adding subqueries to your statements, and creating unions, you're ready to move beyond accessing data in multiple tables and creating statements that access multiple tables in managing data that extends beyond those tables. In Chapter 11, you learn how to export, copy, and import data.

Exercises

For these exercises, you create several `SELECT` statements that use joins, subqueries, and unions. You use the same tables that have been used for the examples throughout the chapter. These include the Books, AuthorBook, Authors, BookOrders, and Authors2 tables (shown in Figure 10-4). Refer back to the table definitions and their related `INSERT` statements as necessary if you have any questions about the tables. You can find the answers to these exercises in Appendix A.

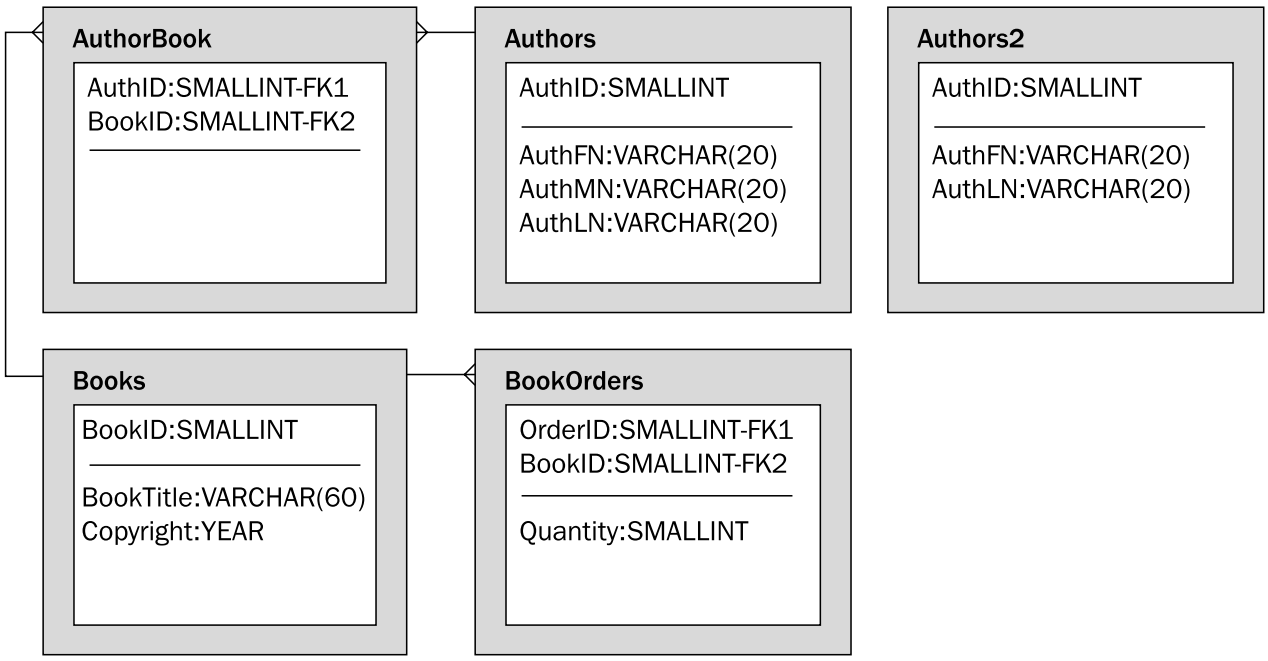

Figure 10-4

1. Create a `SELECT` statement that uses a basic join to retrieve the name of books in the Books table and the name of the authors in the Authors table who wrote those books. The author names should be concatenated with a space between names. In addition, the result set should be sorted according to the book titles.
2. Create a `SELECT` statement that is similar to the statement that you created in Step 1. The new statement should use a cross join to join the tables, and the rows returned should be limited to those books that were written by authors whose last names begin with Toole or Thompson.
3. Create a `SELECT` statement that uses a right outer join to retrieve the name of books in the Books table and the name of the authors in the Authors table who wrote those books. The Authors table should be on the right side of the join. The author names should be concatenated with a space between names. In addition, the result set should be sorted according to the book titles.
4. Create a `SELECT` statement similar to the one that you created in Step 3. The new statement should use a natural right join to join the tables.
5. Create a `SELECT` statement that retrieves book titles from the Books table. The result set should include only those books whose BookID value is equal to one of the values returned by a subquery. The subquery should return BookID values from the BookOrders table for those books that sold more than two books in an order. The result set should be sorted according to the book titles.
6. Create a `SELECT` statement that retrieves OrderID, BookID, and Quantity values from the BookOrders table for the book *Letters to a Young Poet*. The statement should use a subquery to retrieve the BookID value associated with the book.
7. Use a `UNION` operator to join two `SELECT` statements. The first `SELECT` statement should retrieve the AuthLN column from the Authors table. The second `SELECT` statement should retrieve the AuthLN column from the Authors2 table. The names returned by the statement should be sorted alphabetically, in ascending order.