7

Retrieving Data from a MySQL Database

One of the most important functions that a relational database management system (RDBMS) must support is the ability to access data in the databases managed by that system. Data access must extend beyond the mere retrieval of information as it is stored in the tables. You must be able to choose which data you want to view and how that data is displayed. To support this functionality, MySQL provides an SQL statement that is both powerful and flexible in its implementation. The `SELECT` statement is the primary SQL statement used in MySQL — and in most RDBMSs — to retrieve specific data from one or more tables in a relational database.

By using a `SELECT` statement, you can specify which columns and which rows to retrieve from one or more tables in your MySQL database. You can also link values together across multiple tables, perform calculations on those values, or group values together in meaningful ways in order to provide summarized information. When you execute a `SELECT` statement, the values returned by that statement are presented in the form of a *result set*, which is an unnamed temporary table that contains the information retrieved from the tables. In this chapter, you will learn how to create `SELECT` statements that allow you to retrieve exactly the information that you need. Specifically, the chapter covers the following topics:

- The SELECT statement and its syntax, as used in MySQL. You learn how to create statements that retrieve all columns in a table or only specific columns. You also learn how to add expressions and define variables in your `SELECT` statements.
- How to add options to a `SELECT` statement that determine how the statement is executed.
- How to add optional clauses to your `SELECT` statement that allow you to limit which rows are returned, to group together rows in order to summarize information, and to specify the order in which rows are displayed.

The SELECT Statement

Whenever you want to retrieve data from a MySQL database, you can issue a SELECT statement that specifies what data you want to have returned and in what manner that data should be returned. For example, you can specify that only specific columns or rows be returned. You can also order the rows based on the values in one or more columns. In addition, you can group together rows based on repeated values in a column in order to summarize data.

The SELECT statement is one of the most powerful SQL statements in MySQL. It provides a great deal of flexibility and allows you to create queries that are as simple or as complex as you need to make them. The syntax for a SELECT statement is made up of a number of clauses and other elements, most of which are optional, that allow you to refine your query so that it returns only the information that you're looking for. The following syntax describes the elements that make up a SELECT statement:

```
<select statement>::=
SELECT
[<select option> [<select option>...]]
{* | <select list>}
[<export definition>]
[
   FROM <table reference> [{, <table reference>}...]
   [WHERE <expression> [{<operator> <expression>}...]]
   [GROUP BY <group by definition>]
   [HAVING <expression> [{<operator> <expression>}...]]
   [ORDER BY <order by definition>]
   [LIMIT [<offset>,] <row count>]
   [PROCEDURE <procedure name> [(<argument> [{, <argument>}...])]]
   [{FOR UPDATE} | {LOCK IN SHARE MODE}]
]

<select option>::=
{ALL | DISTINCT | DISTINCTROW}
| HIGH_PRIORITY
| {SQL_BIG_RESULT | SQL_SMALL_RESULT}
| SQL_BUFFER_RESULT
| {SQL_CACHE | SQL_NO_CACHE}
| SQL_CALC_FOUND_ROWS
| STRAIGHT_JOIN

<select list>::=
{<column name> | <expression>} [[AS] <alias>]
[{, {<column name> | <expression>} [[AS] <alias>]}...]

<export definition>::=
INTO OUTFILE '<filename>' [<export option> [<export option>]]
| INTO DUMPFILE '<filename>'

<export option>::=
{FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']}
| {LINES
```

```
    [STARTING BY '<value>']
     [TERMINATED BY '<value>']}

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

<group by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
[WITH ROLLUP]

<order by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
```

As you can see from the syntax, a `SELECT` statement can contain a number of elements. For most of these elements, the chapter discusses each one in detail, providing the necessary examples to illustrate how they work; however, some elements are covered in later chapters. For example, the `<export definition>` option is discussed in Chapter 11, and the `FOR UPDATE` and the `LOCK IN SHARE MODE` options are discussed in Chapter 12. In addition, the `PROCEDURE` clause, which is used to call C++ procedures, is not discussed; it is beyond the scope of the book. It is included here only to provide you with the complete `SELECT` statement syntax.

Referring back to the syntax, notice that a `SELECT` syntax requires only the following clause:

```
SELECT
{* | <select list>}
```

The `SELECT` clause includes the `SELECT` keyword and an asterisk (*) or the select list, which is made up of columns or expressions, as shown in the following syntax:

```
<select list>::=
{<column name> | <expression>} [[AS] <alias>]
[{, {<column name> | <expression>} [[AS] <alias>]}...]
```

As you can see, the select list must include at least one column name or one expression. If more than one column/expression element is included, they must be separated by commas. In addition, you can assign an alias to a column name or expression by using the `AS` subclause. That alias can be then be used in other clauses in the `SELECT` statement; however, it cannot be used in a `WHERE` clause because of the way in which MySQL processes a `SELECT` statement.

As the syntax indicates, the `AS` keyword is optional when assigning an alias to a column. For the sake of clarity and to avoid confusion with other column names, it is generally recommended that you include the `AS` keyword.

Although the `SELECT` clause is the only required element in a `SELECT` statement, you cannot retrieve data from a table unless you also specify a `FROM` clause and the appropriate table references. The `FROM` clause requires one or more table references, separated by commas, as shown in the following syntax:

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

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

Each table reference is made up of a table name and an optional AS subclause that allows you to assign an alias to a table. The FROM clause can include more than one table reference; however, multiple references are included only if you are joining two or more tables in your SELECT statement. In addition, you would generally assign a table alias or include a {USE | IGNORE | FORCE} INDEX clause only when joining a table. (Table joins are discussed in detail in Chapter 10.) When creating a SELECT statement that does not include joined tables, your FROM clause normally includes only the FROM keyword and the name of the target table.

As the syntax shows, creating a basic SELECT statement that retrieves data from only one table requires few components. To demonstrate how this works, take a look at a few examples. The examples in this chapter are all based on a table named CDs, which is shown in the following table definition:

```
CREATE TABLE CDs
(
   CDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
   CDName VARCHAR(50) NOT NULL,
   InStock SMALLINT UNSIGNED NOT NULL,
   OnOrder SMALLINT UNSIGNED NOT NULL,
   Reserved SMALLINT UNSIGNED NOT NULL,
   Department ENUM('Classical', 'Popular') NOT NULL,
   Category VARCHAR(20) NOT NULL,
   RowUpdate TIMESTAMP NOT NULL
);
```

For the purposes of this chapter, you can assume that the following INSERT statement has been used to add data to the CDs table:

```
INSERT INTO CDs (CDName, InStock, OnOrder, Reserved, Department, Category)
VALUES ('Bloodshot', 10, 5, 3, 'Popular', 'Rock'),
('The Most Favorite Opera Duets', 10, 5, 3, 'Classical', 'Opera'),
('New Orleans Jazz', 17, 4, 1, 'Popular', 'Jazz'),
('Music for Ballet Class', 9, 4, 2, 'Classical', 'Dance'),
('Music for Solo Violin', 24, 2, 5, 'Classical', 'General'),
('Cie li di Toscana', 16, 6, 8, 'Classical', 'Vocal'),
('Mississippi Blues', 2, 25, 6, 'Popular', 'Blues'),
('Pure', 32, 3, 10, 'Popular', 'Jazz'),
('Mud on the Tires', 12, 15, 13, 'Popular', 'Country'),
('The Essence', 5, 20, 10, 'Popular', 'New Age'),
('Embrace', 24, 11, 14, 'Popular', 'New Age'),
('The Magic of Satie', 42, 17, 17, 'Classical', 'General'),
('Swan Lake', 25, 44, 28, 'Classical', 'Dance'),
('25 Classical Favorites', 32, 15, 12, 'Classical', 'General'),
('La Boheme', 20, 10, 5, 'Classical', 'Opera'),
('Bach Cantatas', 23, 12, 8, 'Classical', 'General'),
('Golden Road', 23, 10, 17, 'Popular', 'Country'),
('Live in Paris', 18, 20, 10, 'Popular', 'Jazz'),
('Richland Woman Blues', 22, 5, 7, 'Popular', 'Blues'),
    ('Morimur (after J. S. Bach)', 28, 17, 16, 'Classical', 'General'),
    ('The Best of Italian Opera', 10, 35, 12, 'Classical', 'Opera'),
    ('Runaway Soul', 15, 30, 14, 'Popular', 'Blues'),
    ('Stages', 42, 0, 8, 'Popular', 'Blues'),
    ('Bach: Six Unaccompanied Cello Suites', 16, 8, 8, 'Classical', 'General');
```

As you work your way through the chapter, you might want to refer to this `CREATE TABLE` statement and `INSERT` statement to reference the table and its contents as example `SELECT` statements are presented. Now take a look at a `SELECT` statement that accesses data in the CDs table:

```
SELECT * FROM CDs;
```

The statement includes a `SELECT` clause and a `FROM` clause. The `SELECT` clause includes the `SELECT` keyword and an asterisk. The asterisk indicates that the query should return all columns. The `FROM` clause includes the `FROM` keyword and the name of the CDs table. When you execute this statement, you should receive results similar to the following:

```
+------+--------------------------------------+---------+---------+----------+-----
| CDID | CDName                               | InStock | OnOrder | Reserved | Depa
+------+--------------------------------------+---------+---------+----------+-----
|    1 | Bloodshot                            |      10 |       5 |        3 | Popu
|    2 | The Most Favorite Opera Duets        |      10 |       5 |        3 | Clas
|    3 | New Orleans Jazz                     |      17 |       4 |        1 | Popu
|    4 | Music for Ballet Class               |       9 |       4 |        2 | Clas
|    5 | Music for Solo Violin                |      24 |       2 |        5 | Clas
|    6 | Cie li di Toscana                    |      16 |       6 |        8 | Clas
|    7 | Mississippi Blues                    |       2 |      25 |        6 | Popu
|    8 | Pure                                 |      32 |       3 |       10 | Popu
|    9 | Mud on the Tires                     |      12 |      15 |       13 | Popu
|   10 | The Essence                          |       5 |      20 |       10 | Popu
|   11 | Embrace                              |      24 |      11 |       14 | Popu
|   12 | The Magic of Satie                   |      42 |      17 |       17 | Clas
|   13 | Swan Lake                            |      25 |      44 |       28 | Clas
|   14 | 25 Classical Favorites               |      32 |      15 |       12 | Clas
|   15 | La Boheme                            |      20 |      10 |        5 | Clas
|   16 | Bach Cantatas                        |      23 |      12 |        8 | Clas
|   17 | Golden Road                          |      23 |      10 |       17 | Popu
|   18 | Live in Paris                        |      18 |      20 |       10 | Popu
|   19 | Richland Woman Blues                 |      22 |       5 |        7 | Popu
|   20 | Morimur (after J. S. Bach)           |      28 |      17 |       16 | Clas
|   21 | The Best of Italian Opera            |      10 |      35 |       12 | Clas
|   22 | Runaway Soul                         |      15 |      30 |       14 | Popu
|   23 | Stages                               |      42 |       0 |        8 | Popu
|   24 | Bach: Six Unaccompanied Cello Suites |      16 |       8 |        8 | Clas
+------+--------------------------------------+---------+---------+----------+-----
24 rows in set (0.00 sec)
```

Because the rows are so long, only a part of each row is displayed here. How these rows would appear on your system varies; however, the basic components should be the same. The important point to remember is that a `SELECT` clause that contains only an asterisk returns all columns in the table. In addition, all rows are returned as well because the `SELECT` statement has not been qualified in any way other than adding the optional `FROM` clause, which defines the table to access.

Although using an asterisk in the SELECT clause is an easy way to retrieve every column from a table, it is not a recommended method to use when embedding a SELECT statement in a programming language. Columns can change or be added or deleted from a table. Consequently, unless you're simply performing an ad hoc query and want to view a table's contents quickly, you should normally specify the column names, as shown in the following SELECT statement:

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

Notice that, in this case, the query specifies three column names in the SELECT clause: CDID, CDName, and Category. Because these names are specified, only data from these three columns is returned by your query, as shown in the following results:

```
+------+--------------------------------------+----------+
| CDID | CDName                               | Category |
+------+--------------------------------------+----------+
|    1 | Bloodshot                            | Rock     |
|    2 | The Most Favorite Opera Duets        | Opera    |
|    3 | New Orleans Jazz                     | Jazz     |
|    4 | Music for Ballet Class               | Dance    |
|    5 | Music for Solo Violin                | General  |
|    6 | Cie li di Toscana                    | Vocal    |
|    7 | Mississippi Blues                    | Blues    |
|    8 | Pure                                 | Jazz     |
|    9 | Mud on the Tires                     | Country  |
|   10 | The Essence                          | New Age  |
|   11 | Embrace                              | New Age  |
|   12 | The Magic of Satie                   | General  |
|   13 | Swan Lake                            | Dance    |
|   14 | 25 Classical Favorites               | General  |
|   15 | La Boheme                            | Opera    |
|   16 | Bach Cantatas                        | General  |
|   17 | Golden Road                          | Country  |
|   18 | Live in Paris                        | Jazz     |
|   19 | Richland Woman Blues                 | Blues    |
|   20 | Morimur (after J. S. Bach)           | General  |
|   21 | The Best of Italian Opera            | Opera    |
|   22 | Runaway Soul                         | Blues    |
|   23 | Stages                               | Blues    |
|   24 | Bach: Six Unaccompanied Cello Suites | General  |
+------+--------------------------------------+----------+
24 rows in set (0.01 sec)
```

Notice that the same number of rows are returned here as were returned in the previous example; however, only three columns of data are displayed. Later in the chapter, you learn how to use the other optional clauses in the SELECT statement to refine your query even further. For now, return to the SELECT clause. If you refer to the select list syntax, you notice that for each element in the select list you can assign an alias, as shown in the following SELECT statement:

```
SELECT CDName AS Title, OnOrder AS Ordered
FROM CDs;
```

Notice that the CDName column is assigned the alias Title and the OnOrder column is assigned the alias Ordered. When you execute this statement, the alias names are used as column headings in your query results, as shown in the following results:

```
+-------------------------------------+---------+
| Title                               | Ordered |
+-------------------------------------+---------+
| Bloodshot                           |       5 |
| The Most Favorite Opera Duets       |       5 |
| New Orleans Jazz                    |       4 |
| Music for Ballet Class              |       4 |
| Music for Solo Violin               |       2 |
| Cie li di Toscana                   |       6 |
| Mississippi Blues                   |      25 |
| Pure                                |       3 |
| Mud on the Tires                    |      15 |
| The Essence                         |      20 |
| Embrace                             |      11 |
| The Magic of Satie                  |      17 |
| Swan Lake                           |      44 |
| 25 Classical Favorites              |      15 |
| La Boheme                           |      10 |
| Bach Cantatas                       |      12 |
| Golden Road                         |      10 |
| Live in Paris                       |      20 |
| Richland Woman Blues                |       5 |
| Morimur (after J. S. Bach)          |      17 |
| The Best of Italian Opera           |      35 |
| Runaway Soul                        |      30 |
| Stages                              |       0 |
| Bach: Six Unaccompanied Cello Suites |       8 |
+-------------------------------------+---------+
24 rows in set (0.02 sec)
```

In a situation like this, in which column names are used rather than expressions and the column names are short and simple, supplying an alias isn't particularly beneficial. As you add expressions to your SELECT clause, create join conditions, or decide to clarify column names, aliases become very useful.

Once you've mastered the SELECT clause and the FROM clause, you can create a basic SELECT statement to query data in any table. In most cases, however, you want to limit the number of rows returned and control how data is displayed. For this, you need to add more clauses. These clauses, which are explained throughout the rest of the chapter, must be added to your statement in the order they are listed in the syntax. MySQL processes the clauses in a SELECT statement in a very specific order, so you must be aware of how clauses are defined in order to receive the results that you expect.

In the following Try It Out exercise, you create three SELECT statements, each of which retrieves data from the Employees table in the DVDRentals database.

Try It Out Creating a SELECT Statement

The following steps describe how to create these statements:

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

```
use DVDRentals
```

   You should receive a message indicating the switch to the DVDRentals database.

2. The first SELECT statement that you create retrieves all columns and all records from the Employees table. To retrieve the records, execute the following SELECT statement at the mysql command prompt:

```
SELECT * FROM Employees;
```

   You should receive results similar to the following:

```
+-------+--------+-------+-----------+
| EmpID | EmpFN  | EmpMN | EmpLN     |
+-------+--------+-------+-----------+
|     1 | John   | P.    | Smith     |
|     2 | Robert | NULL  | Schroader |
|     3 | Mary   | Marie | Michaels  |
|     4 | John   | NULL  | Laguci    |
|     5 | Rita   | C.    | Carter    |
|     6 | George | NULL  | Brooks    |
+-------+--------+-------+-----------+
6 rows in set (0.23 sec)
```

3. Next, retrieve values only from the EmpFN and EmpLN columns of the Employees table. To retrieve the values, execute the following SELECT statement at the mysql command prompt:

```
SELECT EmpFN, EmpLN
FROM Employees;
```

   You should receive results similar to the following:

```
+--------+-----------+
| EmpFN  | EmpLN     |
+--------+-----------+
| John   | Smith     |
| Robert | Schroader |
| Mary   | Michaels  |
| John   | Laguci    |
| Rita   | Carter    |
| George | Brooks    |
+--------+-----------+
6 rows in set (0.00 sec)
```

4. Now retrieve values from the same columns as the last step, but this time provide aliases for those columns. To retrieve the values, execute the following SELECT statement at the mysql command prompt:

```
SELECT EmpFN AS 'First Name', EmpLN AS 'Last Name'
FROM Employees;
```

You should receive results similar to the following:

```
+------------+-----------+
| First Name | Last Name |
+------------+-----------+
| John       | Smith     |
| Robert     | Schroader |
| Mary       | Michaels  |
| John       | Laguci    |
| Rita       | Carter    |
| George     | Brooks    |
+------------+-----------+
6 rows in set (0.01 sec)
```

How It Works

Whenever you create a SELECT statement that retrieves data from a table in a MySQL database, you must, at the very least, include a SELECT clause and a FROM clause. The SELECT clause determines which columns of values are returned and the FROM clause determines from which tables the data is retrieved. For example, the first SELECT statement that you created retrieves all columns from the Employees table, as shown in the following statement:

```
SELECT * FROM Employees;
```

This statement uses an asterisk to indicate that all columns should be retrieved. Your second SELECT statement specified which columns of data should be returned:

```
SELECT EmpFN, EmpLN
FROM Employees;
```

In this case, the query returns only values in the EmpFN and EmpLN columns because those are the columns specified in the SELECT clause. And as with the first SELECT statement in this exercise, the values were retrieved from the Employees table because that is the table specified in the FROM clause.

The last SELECT statement that you created in this exercise assigns aliases to the columns names, as shown in the following statement:

```
SELECT EmpFN AS 'First Name', EmpLN AS 'Last Name'
FROM Employees;
```

Notice that you assigned the EmpFN column the alias First Name and that you assigned the EmpLN column the alias Last Name. In both cases, you enclosed the aliases in a single quote because they were each made up of more than one word and the quotations were necessary to ensure that both words were considered part of the alias name.

Using Expressions in a SELECT Statement

Recalling from the select list syntax, your select list can include column names or expressions. Up to this point, the example SELECT statements that you've seen have included columns names. Expressions are also very useful in creating robust SELECT statements that can return the data necessary to your applications.

An expression, as you learned in Chapter 6, is a type of formula that helps define the value that the SELECT statement will return. An expression can include column names, literal values, operators, and functions. An operator is a symbol that represents the action that should be taken, such as comparing values or adding values together. For example, the minus (-) sign is an arithmetic operator that is used to subtract one value from another. Another example is the greater than (>) operator, which is a comparison operator used to compare values to determine if one value is greater than the other. (A comparison operator is a type of operator that compares two values.) A function is an object that carries out a predefined task. For example, you can use a function to specify the current date.

Although Chapter 8 discusses operators in greater detail and Chapter 9 discusses functions, both operators and functions are often an integral part of an expression, so some operators and functions are included here so that you can better understand how to use an expression in a SELECT clause. Keep in mind, however, that after you've learned more about operators and functions in Chapters 8 and 9, you will be able to create even more robust expressions in your SELECT clause or anywhere else that you can use expressions.

Now take a look at a SELECT statement that contains an expression in its select list. The statement is based on the table definition for the CDs table that you saw earlier in the chapter. The following statement retrieves information from several columns in the table:

```
SELECT CDName, InStock+OnOrder AS Total
FROM CDs;
```

The first element of the select list is the CDName column, which is included here in the same way that you've seen other column names added to a select list. The second element is an expression, though, rather than a column name. The expression (`InStock+OnOrder`) adds together the values from the InStock and OnOrder columns for each row returned by the SELECT statement. The expression is assigned the name Total, which is the name used in the query results to display the values, as the following results show:

```
+--------------------------------------+-------+
| CDName                               | Total |
+--------------------------------------+-------+
| Bloodshot                            |    15 |
| The Most Favorite Opera Duets        |    15 |
| New Orleans Jazz                     |    21 |
| Music for Ballet Class               |    13 |
| Music for Solo Violin                |    26 |
| Cie li di Toscana                    |    22 |
| Mississippi Blues                    |    27 |
| Pure                                 |    35 |
| Mud on the Tires                     |    27 |
| The Essence                          |    25 |
| Embrace                              |    35 |
| The Magic of Satie                   |    59 |
| Swan Lake                            |    69 |
| 25 Classical Favorites              |    47 |
| La Boheme                           |    30 |
| Bach Cantatas                       |    35 |
| Golden Road                         |    33 |
| Live in Paris                       |    38 |
| Richland Woman Blues                |    27 |
| Morimur (after J. S. Bach)          |    45 |
| The Best of Italian Opera           |    45 |
| Runaway Soul                        |    45 |
| Stages                              |    42 |
| Bach: Six Unaccompanied Cello Suites |    24 |
+-------------------------------------+-------+
24 rows in set (0.00 sec)
```

As you can see, the query returns two columns: the CDName column and the Total column. The values in the Total column are based on the expression defined in the `SELECT` clause. If you refer back to the original values that you added to the table, you would see that these values are based on the total from the two columns. For example, in the first row, the CDName value is Bloodshot and the Total value is 15. Currently in the CDs table, the InStock value is 10 and the OnOrder value is 5. When added together, the total is 15, which is the amount inserted in the Total column of the result set.

You can also create expressions in your select list that are more complex than the one in the last statement. For example, the following `SELECT` statement is similar to the last, except that the expression in the select list now subtracts the Reserved value from the total:

```
SELECT CDName, InStock+OnOrder-Reserved AS Total
FROM CDs;
```

To arrive at the value in the Total column, the InStock and OnOrder values are added together, and then the Reserved value is subtracted from this total, as shown in the following results:

```
+-------------------------------------+-------+
| CDName                              | Total |
+-------------------------------------+-------+
| Bloodshot                           |    12 |
| The Most Favorite Opera Duets       |    12 |
| New Orleans Jazz                    |    20 |
| Music for Ballet Class              |    11 |
| Music for Solo Violin               |    21 |
| Cie li di Toscana                   |    14 |
| Mississippi Blues                   |    21 |
| Pure                                |    25 |
| Mud on the Tires                    |    14 |
| The Essence                         |    15 |
| Embrace                             |    21 |
| The Magic of Satie                  |    42 |
| Swan Lake                           |    41 |
| 25 Classical Favorites              |    35 |
| La Boheme                           |    25 |
| Bach Cantatas                       |    27 |
| Golden Road                         |    16 |
| Live in Paris                       |    28 |
| Richland Woman Blues                 |    20 |
| Morimur (after J. S. Bach)           |    29 |
| The Best of Italian Opera            |    33 |
| Runaway Soul                         |    31 |
| Stages                               |    34 |
| Bach: Six Unaccompanied Cello Suites |    16 |
+--------------------------------------+-------+
24 rows in set (0.00 sec)
```

For each row that the SELECT statement returns, the expression is calculated and the result inserted in the Total column. When you use arithmetic operators in an expression, the components of that expression are evaluated according to the basic formulaic principles of mathematics. In Chapter 8, where operators are discussed in greater detail, you learn more about how an expression is evaluated based on the operators used in that expression.

As you have seen, you can specify column names or expressions in the SELECT clause. In the following exercise, you create a SELECT statement that includes an expression.

Try It Out Adding Expressions to Your Select List

Follow these steps to add expressions to your select list:

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

```
use DVDRentals
```

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

2. To retrieve the records from the Employees table, execute the following SELECT statement at the mysql command prompt:

```
SELECT EmpID, EmpFN, EmpLN
FROM Employees;
```

   You should receive results similar to the following:

```
+-------+--------+-----------+
| EmpID | EmpFN  | EmpLN     |
+-------+--------+-----------+
|     1 | John   | Smith     |
|     2 | Robert | Schroader |
|     3 | Mary   | Michaels  |
|     4 | John   | Laguci    |
|     5 | Rita   | Carter    |
|     6 | George | Brooks    |
+-------+--------+-----------+
6 rows in set (0.03 sec)
```

How It Works

In this exercise, you created a SELECT statement that included a SELECT clause that contained three select list elements, as shown in the following statement:

```
SELECT EmpID, EmpFN, EmpLN
FROM Employees;
```

The three select list elements consist of the name of columns in the Employees table. Because the columns are included here, the SELECT statement returns values from those columns (as they appear in the Employees table).

Using Variables in a SELECT Statement

One type of expression that you can include in your select list is one that allows you to define a variable. A *variable* is a type of placeholder that holds a value for the duration of a client session. This is useful if you want to reuse a value in later `SELECT` statements.

You define a variable by using the following structure:

```
@<variable name>:={<column name> | <expression>} [[AS] <alias>]
```

The variable name must always be preceded by the at (@) symbol, and the variable value must always be specified by using the colon/equal sign (:=) symbols. In addition, a variable can be associated with only one value, so your `SELECT` statement should return only one value per variable. If your `SELECT` statement returns more than one value for a variable, the last value returned is used by the variable. If you want to define more than one variable in a `SELECT` statement, you must define each one as a separate select list element. For example, the following `SELECT` statement defines two variables:

```
SELECT @dept:=Department, @cat:=Category
FROM CDs
WHERE CDName='Mississippi Blues';
```

When you execute this statement, the values from the Department column and the Category column are stored in the appropriate variables. For example, the row in the CDs table that contains a CDName of Mississippi Blues contains a Department value of Popular and a Category value of Blues. As a result, the Popular value is assigned to the `@dept` variable, and the Blues value is assigned to the `@cat` variable. When you execute the `SELECT` statement, you should receive results similar to the following:

```
+-------------------+----------------+
| @dept:=Department | @cat:=Category |
+-------------------+----------------+
| Popular           | Blues          |
+-------------------+----------------+
1 row in set (0.26 sec)
```

As you can see, your result set displays the values assigned to your variables. Once you've assigned values to your variables, you can then use them in other `SELECT` statements, as shown in the following example:

```
SELECT CDID, CDName, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE Department=@dept AND Category=@cat;
```

The first thing that you might notice in this `SELECT` statement is that it contains a `WHERE` clause. Although `WHERE` clauses are discussed in greater detail later in the chapter — see the section "The WHERE Clause" — they are used here to demonstrate how to use the variables to refine your `SELECT` statement. In this case, the `WHERE` clause includes two conditions. In the first (`Department=@dept`), the condition specifies that any rows returned must contain a Department value that equals the `@dept` variable value, which is Popular. The second condition (`Category=@cat`) specifies that any rows returned must also contain a Category value that equals the `@cat` variable value of Blues. When you execute this statement, you should receive results similar to the following:

```
+------+----------------------+-----------+
| CDID | CDName               | Available |
+------+----------------------+-----------+
|    7 | Mississippi Blues    |        21 |
|   19 | Richland Woman Blues |        20 |
|   22 | Runaway Soul         |        31 |
|   23 | Stages               |        34 |
+------+----------------------+-----------+
4 rows in set (0.07 sec)
```

Notice that four rows are returned. If you refer to the original values that were inserted in the CDs table, you'll find that each of these rows has a Department value of Popular and a Category value of Blues.

Also note that, in addition to a `SELECT` statement, you can use a `SET` statement to define a variable. For example, the following `SET` statement defines the same two variables you saw previously:

```
SET @dept='Popular', @cat='Blues';
```

In this case, rather than setting the variable values based on values returned by a `SELECT` statement, you can specify the values directly, as shown here. You can then use the variables in subsequent `SELECT` statements in your client session, as you would variables defined in a `SELECT` statement. In either case, the variables are usable for only as long as the client session lasts.

In the following Try It Out, you create a `SELECT` statement that defines a variable, and you then use that variable in a second `SELECT` statement.

Try It Out Defining Variables in Your SELECT Statement

To create these statements, 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. You must first define the variable. To define the variable, execute the following `SELECT` statement at the mysql command prompt:

```
SELECT @rating:=RatingID
FROM DVDs
WHERE DVDName='White Christmas';
```

   You should receive results similar to the following:

```
+-------------------+
| @rating:=RatingID |
+-------------------+
| NR                |
+-------------------+
1 row in set (0.00 sec)
```

3. After creating the variable, you can use it in a SELECT statement. To use the variable to retrieve records from the DVDs table, execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDID, DVDName, MTypeID
FROM DVDs
WHERE RatingID=@rating;
```

You should receive results similar to the following:

```
+-------+------------------------------+---------+
| DVDID | DVDName                      | MTypeID |
+-------+------------------------------+---------+
|     1 | White Christmas              | mt16    |
|     4 | The Maltese Falcon           | mt11    |
|     6 | The Rocky Horror Picture Show | mt12   |
|     7 | A Room with a View           | mt11    |
+-------+------------------------------+---------+
4 rows in set (0.00 sec)
```

How It Works

As you have seen, you can use a SELECT statement to define a variable that you can then use in later SELECT statements. In the first SELECT statement that you created in this exercise, you defined the @rating variable, as shown in the following statement:

```
SELECT @rating:=RatingID
FROM DVDs
WHERE DVDName='White Christmas';
```

The @rating variable is assigned a value based on the RatingID value in the DVDs table. Because the WHERE clause specifies that only the row with a DVDName value of White Christmas should be returned, the RatingID value in that row is the one that is assigned to the @rating variable. As a result, @rating is assigned a value of NR. (Had the SELECT statement returned more than one value, the last value returned would have been used by the variable.)

The next SELECT statement uses the @rating variable to specify that only columns that contain a RatingID value equal to the @rating variable value should be returned, as shown in the following SELECT statement:

```
SELECT DVDID, DVDName, MTypeID
FROM DVDs
WHERE RatingID=@rating;
```

Because @rating is assigned a value of NR, this SELECT statement returns only rows that contain a RatingID value of NR.

Using a SELECT Statement to Display Values

When this chapter first introduced you to the SELECT statement syntax, you may have noticed that nearly all elements of the statement are optional. Although your SELECT statements normally include a

FROM clause, along with other optional clauses and options, these elements are all considered optional because you can use a SELECT statement to return values that are not based on data in a table.

When using only the required elements of a SELECT statement, you need to specify only the SELECT keyword and one or more elements of the select list. The select list can contain literal values, operators, and functions, but no column names. For example, the following SELECT statement includes three select list elements:

```
SELECT 1+3, 'CD Inventory', NOW() AS 'Date/Time';
```

The first select list element (1+3) demonstrates how you can use a SELECT statement to perform calculations. The second select list element ('CD Inventory') is a literal value that simply returns the string enclosed in the single quotes. The third select list element (NOW() AS 'Date/Time') uses a NOW() function to return the current date and time. The element also includes an AS subclause that assigns the name Date/Time to the column returned by the statement. If you execute the SELECT statement, you should receive results similar to the following:

```
+-----+--------------+---------------------+
| 1+3 | CD Inventory | Date/Time           |
+-----+--------------+---------------------+
|   4 | CD Inventory | 2004-08-24 11:39:40 |
+-----+--------------+---------------------+
1 row in set (0.00 sec)
```

As you can see, the first select list element is calculated and a value is returned, the second element returns the literal value, and the third element returns the date and time. Notice that the names of the first two columns in the results set are based on the select list elements because you assigned no alias to those expressions.

As you have seen, you can use a SELECT statement to return values that are not tied to a specific table. The following Try It Out describes how to create SELECT statements that return values.

Try It Out Returning Values from a SELECT Statement

Follow these steps to create a SELECT statement that returns values:

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 includes one expression. Execute the following SELECT statement at the mysql command prompt:

```
SELECT 1+8+2;
```

You should receive results similar to the following:

```
+-------+
| 1+8+2 |
+-------+
|    11 |
+-------+
1 row in set (0.00 sec)
```

3. The next SELECT statement includes two expressions. Execute the following SELECT statement at the mysql command prompt:

```
SELECT 'DVDRentals Database', NOW();
```

You should receive results similar to the following:

```
+---------------------+---------------------+
| DVDRentals Database | NOW()               |
+---------------------+---------------------+
| DVDRentals Database | 2004-08-25 14:07:07 |
+---------------------+---------------------+
1 row in set (0.03 sec)
```

How It Works

In this exercise, you created two SELECT statements that return values other than data contained in a table. The first table simply added several numbers together to produce a total, as shown in the following statement:

```
SELECT 1+8+2;
```

As you can see, you needed to specify only the SELECT keyword and the select list element, which added the numbers 1, 8, and 2 together. When you executed the statement, you should have received a total value of 11.

The next SELECT statement that you executed included two select list elements, as the following statement shows:

```
SELECT 'DVDRentals Database', NOW();
```

The first select list element was a literal value enclosed in single quotes. The second element was the NOW() function. When you executed the SELECT statement, the results included the literal string value of DVDRentals Database and the current date and time.

The SELECT Statement Options

When you create a SELECT statement, your SELECT clause can include one or more options that are specified before the select list. The options define how a SELECT statement is processed and, for the most part, how it applies to the statement as a whole, rather to the specific data returned. As the following syntax shows, you can include a number of options in a SELECT statement:

```
<select option>::=
{ALL | DISTINCT | DISTINCTROW}
| HIGH_PRIORITY
| {SQL_BIG_RESULT | SQL_SMALL_RESULT}
| SQL_BUFFER_RESULT
| {SQL_CACHE | SQL_NO_CACHE}
| SQL_CALC_FOUND_ROWS
| STRAIGHT_JOIN
```

The following table describes each of the options that you can include in a SELECT statement.

Option	Description
ALL DISTINCT DISTINCTROW	The ALL option specifies that a query should return all rows, even if there are duplicate rows. The DIS-TINCT and DISTINCTROW options, which have the same meaning in MySQL, specify that duplicate rows should not be included in the result set. If neither option is specified, ALL is assumed.
HIGH_PRIORITY	The HIGH_PRIORITY option prioritizes the SELECT statement over statements that write data to the target table. Use this option only for SELECT statements that you know will execute quickly.
SQL_BIG_RESULT SQL_SMALL_RESULT	The SQL_BIG_RESULT option informs the MySQL optimizer that the result set will include a large number of rows, which helps the optimizer to process the query more efficiently. The SQL_SMALL_RESULT option informs the MySQL optimizer that the result set will include a small number of rows.
SQL_BUFFER_RESULT	The SQL_BUFFER_RESULT option tells MySQL to place the query results in a temporary table in order to release table locks sooner than they would normally be released. This option is particularly useful for large result sets that take a long time to return to the client.
SQL_CACHE SQL_NO_CACHE	The SQL_CACHE option tells MySQL to cache the query results if the cache is operating in demand mode. The SQL_NO_CACHE option tells MySQL not to cache the query results.
SQL_CALC_FOUND_ROWS	You use the SQL_CALC_FOUND_ROWS option in conjunction with the LIMIT clause. The option specifies what the row count of a result set would be if the LIMIT clause were not used.
STRAIGHT_JOIN	You use the STRAIGHT_JOIN option when joining tables in a SELECT statement. The option tells the optimizer to join the tables in the order specified in the FROM clause. You should use this option to speed up a query if you think that the optimizer is not joining the tables efficiently.

To specify an option in a SELECT statement, you must add it after the SELECT keyword, as shown in the following SELECT statement:

```
SELECT ALL Department, Category
FROM CDs;
```

This statement uses the ALL option to specify that all rows should be included in the result set, even if there are duplicates. The select list follows the ALL keyword. The list includes the Department and

Category columns of the CDs table. As a result, all rows are returned from these two columns, as shown in the following results:

```
+------------+----------+
| Department | Category |
+------------+----------+
| Popular    | Rock     |
| Classical  | Opera    |
| Popular    | Jazz     |
| Classical  | Dance    |
| Classical  | General  |
| Classical  | Vocal    |
| Popular    | Blues    |
| Popular    | Jazz     |
| Popular    | Country  |
| Popular    | New Age  |
| Popular    | New Age  |
| Classical  | General  |
| Classical  | Dance    |
| Classical  | General  |
| Classical  | Opera    |
| Classical  | General  |
| Popular    | Country  |
| Popular    | Jazz     |
| Popular    | Blues    |
| Classical  | General  |
| Classical  | Opera    |
| Popular    | Blues    |
| Popular    | Blues    |
| Classical  | General  |
+------------+----------+
24 rows in set (0.00 sec)
```

As you can see, there are duplicate rows in the result set. For example, there are a number of rows that contain a Department value of Popular and a Category value of Blues. You can eliminate these duplicates by using the `DISTINCT` option, rather then the `ALL` option, as shown in the following statement:

```
SELECT DISTINCT Department, Category
FROM CDs;
```

As you can see, this `SELECT` statement is identical to the previous statement except for the `DISTINCT` option. If you execute this statement, you should receive results similar to the following:

```
+------------+----------+
| Department | Category |
+------------+----------+
| Popular    | Rock     |
| Classical  | Opera    |
| Popular    | Jazz     |
| Classical  | Dance    |
| Classical  | General  |
| Classical  | Vocal    |
| Popular    | Blues    |
| Popular    | Country  |
| Popular    | New Age  |
+------------+----------+
9 rows in set (0.02 sec)
```

Notice that only 9 rows are returned (as opposed to 24 rows in the previous statement). Also notice that the result set no longer contains duplicate rows. Although values are repeated in the Department column, no rows, when the values are taken as a whole, are repeated.

You can also specify multiple options in your SELECT clause, as the following example shows:

```
SELECT DISTINCT HIGH_PRIORITY Department, Category
FROM CDs;
```

Notice that the SELECT clause includes the DISTINCT option and the HIGH_PRIORITY option. Because the HIGH_PRIORITY option has no impact on the values returned, your result set looks the same as the result set in the previous example.

As you have seen, you can add one or more options to your SELECT statement that define the behavior of that statement. For this exercise, you use the ALL and DISTINCT options to return data from the DVDs table of the DVDRentals database.

Try It Out Adding Options to Your SELECT Statement

To create the necessary SELECT statement, use the following steps:

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

```
use DVDRentals
```

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

2. In the first SELECT statement, you specify the ALL option to retrieve records from the DVDs table. Execute the following SELECT statement at the mysql command prompt:

```
SELECT ALL RatingID, StatID
FROM DVDs;
```

   You should receive results similar to the following:

```
+----------+--------+
| RatingID | StatID |
+----------+--------+
| NR       | s1     |
| G        | s2     |
| PG       | s1     |
| NR       | s2     |
| PG       | s2     |
| NR       | s2     |
| NR       | s1     |
| R        | s2     |
+----------+--------+
8 rows in set (0.00 sec)
```

3. Now execute the same SELECT statement as in the last step, only this time use the DISTINCT option, rather than the ALL option. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DISTINCT RatingID, StatID
FROM DVDs;
```

You should receive results similar to the following:

```
+----------+--------+
| RatingID | StatID |
+----------+--------+
| NR       | s1     |
| G        | s2     |
| PG       | s1     |
| NR       | s2     |
| PG       | s2     |
| R        | s2     |
+----------+--------+
6 rows in set (0.01 sec)
```

How It Works

In this exercise, you created two `SELECT` statements, one that included the `ALL` option and one that included the `DISTINCT` option. The first statement was as follows:

```
SELECT ALL RatingID, StatID
FROM DVDs;
```

By specifying the `ALL` option, all rows were returned, whether or not there were duplicates. Each row was made up of values from the RatingID column and the StatID column. Overall, eight rows were returned. The second statement, however, included the `DISTINCT` keyword rather than the `ALL` keyword, as shown in the following statement.

```
SELECT DISTINCT RatingID, StatID
FROM DVDs;
```

When you executed this statement, only six rows were returned, and no rows were duplicated, although values were duplicated in the individual columns.

The Optional Clauses of a SELECT Statement

As you saw earlier in the chapter, the `SELECT` statement syntax includes a number of optional clauses that help you define which rows your `SELECT` statement returns and how those rows display. Of particular importance to creating an effective `SELECT` statement are the `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, and `LIMIT` clauses. As you learned earlier, any of these clauses that you include in your `SELECT` statement must be defined in the order that they are specified in the syntax. The remaining part of the chapter discusses each of these clauses and explains how they must be defined to include them in your `SELECT` statements.

The WHERE Clause

Earlier in the chapter, you saw how you can use a SELECT clause to identify the columns that a SELECT statement returns and how to use a FROM clause to identify the table from which the data is retrieved. In this section, you look at the WHERE clause, which allows you to specify which rows in your table are returned by your query.

The WHERE clause is made up of one or more conditions that define the parameters of the SELECT statement. Each condition is an expression that can consist of column names, literal values, operators, and functions. The following syntax describes how a WHERE clause is defined:

```
WHERE <expression> [{<operator> <expression>}...]
```

As you can see, a WHERE clause must contain at least one expression that defines which rows the SELECT statement returns. When you specify more than one condition in the WHERE clause, those conditions are connected by an AND or an OR operator. The operators specify which condition or combination of conditions must be met.

Now take a look at an example to help demonstrate how a WHERE clause is defined in a SELECT statement. The following SELECT statement contains a WHERE clause that defines a single condition:

```
SELECT CDName, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE Category='Blues';
```

The WHERE clause indicates that only rows with a Category value of Blues should be returned as part of the result set. Because the SELECT clause specifies the CDName column and an expression that is assigned the alias Available, only those two columns are included in the result set, as shown in the following results:

```
+----------------------+-----------+
| CDName               | Available |
+----------------------+-----------+
| Mississippi Blues    |        21 |
| Richland Woman Blues |        20 |
| Runaway Soul         |        31 |
| Stages               |        34 |
+----------------------+-----------+
4 rows in set (0.01 sec)
```

Although the Category column is not displayed in the result set, each of these rows has a Category value of Blues. Because only four rows in the table have a Category value of Blues, only four rows are returned.

In this last example, the WHERE clause defined only one condition. You can define multiple conditions in a clause, though, as the following example demonstrates:

```
SELECT CDName, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE Category='Blues' AND (InStock+OnOrder-Reserved)>30;
```

As you can see, the WHERE clause first specifies that the Category value for each row must be Blues. The WHERE clause also includes a second condition, which contains an expression that adds the values of the InStock and OnOrder columns and then subtracts the value in the Reserved column. This total is then compared to the literal value of 30. The comparison is based on the greater than (>) comparison operator (covered in greater detail in Chapter 8), which indicates that the value on the left must be greater than the value on the right in order for the condition to evaluate to true. Because the two conditions are connected by an AND operator, both conditions must be true in order for a row to be returned. In other words, for each row, the Category value must equal Blues *and* the calculation derived from InStock, OnOrder, and Reserved columns must equal an amount greater than 30.

As you might have noticed, the SELECT clause includes an expression that calculates the same three columns in the same way as they are calculated in the WHERE clause. In addition, the expression in the SELECT clause is assigned the name Available. You would think that, because the same format is used in the WHERE clause, you should be able to refer to the Available expression in the WHERE clause so that you can simply write the expression as Available>30. Because of the way in which MySQL processes SELECT statements, you cannot use column aliases in the WHERE clause. Consequently, you must write out the expression as it is done here.

Another alternative is to add a HAVING *clause to your* SELECT *statement. In that clause, you can use column aliases.* HAVING *clauses are discussed later in the chapter.*

If you were to execute the SELECT statement, you would receive results similar to the following:

```
+--------------+-----------+
| CDName       | Available |
+--------------+-----------+
| Runaway Soul |        31 |
| Stages       |        34 |
+--------------+-----------+
2 rows in set (0.00 sec)
```

In this case, only two rows are returned. For each row, the Category value equals Blues and the calculation of the three columns returns an amount greater than 30.

The next example SELECT statement is similar to the last, except that it adds an additional condition to the WHERE clause and the calculated columns must have a total greater than 20, rather than 30, as shown in the following statement:

```
SELECT CDName, Category, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE (Category='Blues' OR Category='Jazz')
   AND (InStock+OnOrder-Reserved)>20;
```

As you can see, the WHERE clause now specifies that the Category value can equal Jazz. Because the OR operator is used to connect the first two conditions, either condition can be met in order for a row to be returned. In other words, a row must contain a Category value of Blues, *or* it must contain a Category value of Jazz. In addition, the three calculated columns must include a total greater than 20. When you execute this query, you should receive results similar to the following:

```
+-------------------+----------+-----------+
| CDName            | Category | Available |
+-------------------+----------+-----------+
| Mississippi Blues | Blues    |        21 |
| Pure              | Jazz     |        25 |
| Live in Paris     | Jazz     |        28 |
| Runaway Soul      | Blues    |        31 |
| Stages            | Blues    |        34 |
+-------------------+----------+-----------+
5 rows in set (0.00 sec)
```

For each row included in the results set, the Category value is Blues *or* Jazz, *and* the Available value is greater than 20. One thing to note about the WHERE clause in this statement is that the first two conditions are enclosed in parentheses. This is done to ensure that these conditions are grouped together and processed properly. Without the parentheses, MySQL would interpret this to mean that each row must contain a Category value of Blues, or each row must contain a Category value of Jazz and an Available value greater than 20. This would return a result set that included every row with a Category value of Blues and only those rows that had a Category value of Jazz and an Available value greater than 20. The Available value would apply only to the Jazz rows, not the Blues rows.

In general, when specifying more than two conditions in a WHERE clause, it is often a good idea to use parentheses to make the meaning clear, unless the meaning of the clauses is absolutely certain (for example, if you use three conditions connected by OR operators).

In the following exercise, you create SELECT statements that each contain a WHERE clause that determines which rows are returned from the DVDs table.

Try It Out Defining a WHERE Clause in Your SELECT Statement

To create these statements, 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. The first SELECT statement includes a WHERE clause that contains one condition. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s2';
```

You should receive results similar to the following:

```
+-------------------------------+---------+
| DVDName                       | MTypeID |
+-------------------------------+---------+
| What's Up, Doc?               | mt12    |
| The Maltese Falcon            | mt11    |
| Amadeus                       | mt11    |
| The Rocky Horror Picture Show | mt12    |
| Mash                          | mt12    |
+-------------------------------+---------+
5 rows in set (0.00 sec)
```

3. Now amend the SELECT statement that you created in the previous step to include multiple conditions in the WHERE clause. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s1' OR StatID='s3' OR StatID='s4';
```

You should receive results similar to the following:

```
+--------------------+---------+
| DVDName            | MTypeID |
+--------------------+---------+
| White Christmas    | mt16    |
| Out of Africa      | mt11    |
| A Room with a View | mt11    |
+--------------------+---------+
3 rows in set (0.00 sec)
```

4. The next SELECT statement also includes multiple conditions in the WHERE clause; this time use an OR and an AND operator. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s2' AND (RatingID='NR' OR RatingID='G');
```

You should receive results similar to the following:

```
+-----------------------------+---------+
| DVDName                     | MTypeID |
+-----------------------------+---------+
| What's Up, Doc?             | mt12    |
| The Maltese Falcon          | mt11    |
| The Rocky Horror Picture Show | mt12    |
+-----------------------------+---------+
3 rows in set (0.00 sec)
```

How It Works

In this exercise, you created three SELECT statements that each contained a WHERE clause. In the first statement the WHERE clause included only one condition, as shown in the following statement:

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s2';
```

The condition in this WHERE clause specifies that the query should return only rows with a StatID value of s2. As a result, this query returned only five rows, and each of those rows had a StatID value of s2.

In the next SELECT statement that you created, you included a WHERE clause that specified three conditions, as the following statement shows:

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s1' OR StatID='s3' OR StatID='s4';
```

In this case, you connected the three conditions with OR operators. This means that each row returned by the SELECT statement had to contain a StatID value of s1 *or* s3 *or* s4. Because only three rows contained any one of these values, only those rows were returned.

In the final SELECT statement that you created in this exercise, you also included three conditions in the WHERE clause, as shown in the following statement:

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s2' AND (RatingID='NR' OR RatingID='G');
```

The first WHERE clause condition specifies that each row returned must have a StatID value of S2. In addition, each row must also have a RatingID value of NR *or* a RatingID value of G. Notice that these two conditions are enclosed in parentheses to ensure that MySQL evaluates the conditions correctly. Another way to look at this is that any row returned by this statement must have a StatID value of s2 and a RatingID value of NR, or the row must have a StatID value of s2 and a RatingID value of G. In this case, only three rows met the conditions specified in the WHERE clause.

The GROUP BY Clause

Up to this point in the chapter, the components of the SELECT statement that you have been introduced to mostly have to do with returning values from columns and rows. Even when your SELECT clause included an expression, that expression usually performed some type of operation on the values in a column. The GROUP BY clause is a little different from the other elements in the SELECT statement in the way that it is used to group values and summarize information.

Take a look at the syntax to see how this works:

```
GROUP BY <group by definition>

<group by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
[WITH ROLLUP]
```

The GROUP BY clause includes the GROUP BY keywords and the group by definition. The definition must include at least one column, although it can include more than one. If multiple columns are specified, you must separate them with commas.

When you specify a GROUP BY clause, rows are grouped together according to values in the column or columns specified in the clause. As a result, you should group rows only for those columns that contain repeated values. For example, you would not want to group a primary key column because each value is unique, so the GROUP BY process would have no practical application. Columns that are more general in nature, such as categories or types, make good candidates for GROUP BY operations because you can derive meaningful summary values for these sorts of columns.

After you've seen a few examples of how to use a GROUP BY clause, you'll get a better sense of how this works. Before looking at any examples, return to the syntax. As you can see, for each column that you specify in the GROUP BY clause, you can also specify that the grouped values be returned in ascending order (the ASC option) or descending order (the DESC option). If neither option is specified, the ASC option is assumed.

In addition to a list of the grouped columns, a GROUP BY clause can include the WITH ROLLUP option. This option provides additional rows of summary information, depending on the number of columns specified. The WITH ROLLUP option is best explained through the use of examples. Later in this section, you see how the option can provide additional summary information.

In order to use a GROUP BY clause effectively, you should also include a select list element that contains a function that summarizes the data returned by the SELECT statement. For example, suppose you want to know how many compact disk titles are listed in the CDs table for each category. To find out the number, you can use the following SELECT statement:

```
SELECT Category, COUNT(*) AS Total
FROM CDs
WHERE Department='Popular'
GROUP BY Category;
```

As the example shows, the SELECT clause includes the Category column and the COUNT(*) expression (which is assigned the alias Total). The COUNT() function calculates the number of rows for the specified column or for the table as a whole. Because you use an asterisk instead of a column name, all the rows in the table are counted. If you specify a column name and that column contains NULL values, the rows that contain NULL are not counted. (See Chapter 9 for more information about the COUNT() function and other functions that you can use when working with summarized data.)

After the SELECT clause, the FROM clause specifies the CDs table, and the WHERE clause specifies that only rows with a Department value that equals Popular should be returned. The GROUP BY clause then specifies that the rows should be grouped together according to the values in the Category column. As a result, only one row is returned for each unique Category value, as shown in the following result set:

```
+----------+-------+
| Category | Total |
+----------+-------+
| Blues    |     4 |
| Country  |     2 |
| Jazz     |     3 |
| New Age  |     2 |
| Rock     |     1 |
+----------+-------+
5 rows in set (0.08 sec)
```

As you can see, each Category value is listed only once, and a total is provided for each category. For example, the CDs table contains four Blues compact disks and two Country compact discs. Notice that the values in the Category column are listed in ascending order (alphabetically) because this is the default sort order for grouped columns.

Now take a look at an example that specifies two columns in the GROUP BY clause:

```
SELECT Department, Category, COUNT(*) AS Total
FROM CDs
GROUP BY Department, Category;
```

As you can see, the SELECT clause includes the Department and Category columns and the COUNT(*) expression (assigned the alias Total). The GROUP BY clause also includes the Department and Category columns. As a result, the SELECT statement first groups the result set according to the Department values and then according to the Category column, as shown in the following results:

```
+------------+----------+-------+
| Department | Category | Total |
+------------+----------+-------+
| Classical  | Dance    |     2 |
| Classical  | General  |     6 |
| Classical  | Opera    |     3 |
| Classical  | Vocal    |     1 |
| Popular    | Blues    |     4 |
| Popular    | Country  |     2 |
| Popular    | Jazz     |     3 |
| Popular    | New Age  |     2 |
| Popular    | Rock     |     1 |
+------------+----------+-------+
9 rows in set (0.00 sec)
```

As the result set shows, the Classical department includes four categories, and the Popular department includes five categories. For each category, the number of rows returned by that category is listed in the Total column. For example, the Dance category in the Classical department contains two compact disk titles.

Now take this same statement and add the `WITH ROLLUP` option to the `GROUP BY` clause, as shown in the following example:

```
SELECT Department, Category, COUNT(*) AS Total
FROM CDs
GROUP BY Department, Category WITH ROLLUP;
```

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

```
+------------+----------+-------+
| Department | Category | Total |
+------------+----------+-------+
| Classical  | Dance    |     2 |
| Classical  | General  |     6 |
| Classical  | Opera    |     3 |
| Classical  | Vocal    |     1 |
| Classical  | NULL     |    12 |
| Popular    | Blues    |     4 |
| Popular    | Country  |     2 |
| Popular    | Jazz     |     3 |
| Popular    | New Age  |     2 |
| Popular    | Rock     |     1 |
| Popular    | NULL     |    12 |
| NULL       | NULL     |    24 |
+------------+----------+-------+
12 rows in set (0.00 sec)
```

Notice the several additional rows in the result set. For example, the fifth row (the last Classical entry) includes `NULL` in the Category column and 12 in the Total column. The `WITH ROLLUP` option provides summary data for the first column specified in the `GROUP BY` clause, as well as the second column. As this shows, there are a total of 12 Classical compact disks listed in the CDs table. A summarized value is also provided for the Popular department. There are 12 Popular compact disks as well. The last row in the result set provides a total for all compact disks. As the Total value shows, there are 24 compact disks in all.

The type of summarized data included in your result set depends on the summary functions used in the SELECT clause. MySQL supports a number of summary functions that work in conjunction with the GROUP BY clause. Chapter 9 contains detailed information about these and other functions available in MySQL.

In the following exercise, you create several SELECT statements, each of which includes a GROUP BY clause. The first statement groups values by a single column, and the next two statements group values by two columns.

Try It Out Defining a GROUP BY Clause in Your SELECT Statement

The following steps describe how to create these 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 SELECT statement groups together the values in the OrderID column. Execute the following SELECT statement at the mysql command prompt:

```
SELECT OrderID, COUNT(*) AS Transactions
FROM Transactions
GROUP BY OrderID;
```

You should receive results similar to the following:

```
+---------+--------------+
| OrderID | Transactions |
+---------+--------------+
|       1 |            3 |
|       2 |            1 |
|       3 |            3 |
|       4 |            1 |
|       5 |            1 |
|       6 |            2 |
|       7 |            1 |
|       8 |            3 |
|       9 |            2 |
|      10 |            1 |
|      11 |            3 |
|      12 |            1 |
|      13 |            1 |
+---------+--------------+
13 rows in set (0.01 sec)
```

3. The next SELECT statement groups together values by the MTypeID and RatingID columns. Execute the following SELECT statement at the mysql command prompt:

```
SELECT MTypeID, RatingID, COUNT(*) AS 'DVD Totals'
FROM DVDs
GROUP BY MTypeID, RatingID;
```

You should receive results similar to the following:

```
+---------+----------+------------+
| MTypeID | RatingID | DVD Totals |
+---------+----------+------------+
| 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)
```

4. The third SELECT statement is similar to the one in the last step except that you now include the WITH ROLLUP option in the GROUP BY clause. Execute the following SELECT statement at the mysql command prompt:

```
SELECT MTypeID, RatingID, COUNT(*) AS 'DVD Totals'
FROM DVDs
GROUP BY MTypeID, RatingID WITH ROLLUP;
```

You should receive results similar to the following:

```
+---------+----------+------------+
| MTypeID | RatingID | DVD Totals |
+---------+----------+------------+
| mt11    | NR       |          2 |
| mt11    | PG       |          2 |
| mt11    | NULL     |          4 |
| mt12    | G        |          1 |
| mt12    | NR       |          1 |
| mt12    | R        |          1 |
| mt12    | NULL     |          3 |
| mt16    | NR       |          1 |
| mt16    | NULL     |          1 |
| NULL    | NULL     |          8 |
+---------+----------+------------+
10 rows in set (0.00 sec)
```

How It Works

The first SELECT statement that you created in this exercise includes a GROUP BY column that groups data according to the OrderID column of the Transactions table, as shown in the following statement:

```
SELECT OrderID, COUNT(*) AS Transactions
FROM Transactions
GROUP BY OrderID;
```

The SELECT clause in the statement specifies the OrderID column and the COUNT(*) expression, which is assigned the alias Transactions. The GROUP BY clause then specifies that the result set be grouped together based on the OrderID column. The result set returns the number of transactions per order.

The next SELECT statement that you created groups together the values from two columns, as the following SELECT statement shows:

```
SELECT MTypeID, RatingID, COUNT(*) AS 'DVD Totals'
FROM DVDs
GROUP BY MTypeID, RatingID;
```

In this case, the rows are grouped together first by the values in the MTypeID column and then by the RatingID values. For each MTypeID/RatingID pair, a total number of DVDs is provided. For example, for the MTypeID value of mt11, the RatingID value of NR includes two DVDs, and the RatingID value of PG includes two DVDs.

The last `SELECT` statement that you created is identical to the previous one, except that it includes the `WITH ROLLUP` option in the `GROUP BY` clause, as shown in the following statement:

```
SELECT MTypeID, RatingID, COUNT(*) AS 'DVD Totals'
FROM DVDs
GROUP BY MTypeID, RatingID WITH ROLLUP;
```

The `WITH ROLLUP` option adds rows to the result set that provide additional summary values. In this case, a total is provided for each MTypeID value as well as an overall total.

The HAVING Clause

The `HAVING` clause is very similar to the `WHERE` clause in that it consists of one or more conditions that define which rows are included in a result set. The `HAVING` clause, though, has a couple of advantages over the `WHERE` clause. For example, you can include aggregate functions in a `HAVING` clause. An *aggregate function* is a type of function that summarizes data, such as the `COUNT()` function. You cannot use aggregate functions in expressions in your `WHERE` clause. In addition, you can use column aliases in a `HAVING` clause, which you cannot do in a `WHERE` clause.

Despite the disadvantages of the `WHERE` clause, whenever an expression can be defined in either a `HAVING` clause or a `WHERE` clause, it is best to use the `WHERE` clause because of the way that MySQL optimizes queries. In general, the `HAVING` clause is normally best suited to use in conjunction with the `GROUP BY` clause.

To include a `HAVING` clause in your `SELECT` statement, you must include the `HAVING` keyword and at least one expression, as shown in the following syntax:

```
HAVING <expression> [{<operator> <expression>}...]
```

A `HAVING` clause is constructed exactly like a `WHERE` clause, in terms of defining conditions and connecting multiple conditions with operators. For example, the following `SELECT` statement includes a `HAVING` clause that contains one condition:

```
SELECT Category, COUNT(*) AS Total
FROM CDs
WHERE Department='Popular'
GROUP BY Category
HAVING Total<3;
```

You should be familiar with most of the elements in this statement. The `SELECT` clause includes the name of the Category column and the expression `COUNT(*)`, which summarizes the grouped data. The `WHERE` clause specifies that all returned rows must have a Department value of Popular, and the `GROUP BY` clause specifies that the rows should be grouped together based on the values in the Category column.

The HAVING clause adds another element to the SELECT statement by specifying that the value in the Total column in the result set must be less than 3. The Total column in the result set shows the values that are returned by the COUNT(*) expression defined in the SELECT clause. The SELECT statement should return the following results:

```
+----------+-------+
| Category | Total |
+----------+-------+
| Country  |     2 |
| New Age  |     2 |
| Rock     |     1 |
+----------+-------+
3 rows in set (0.00 sec)
```

The result set includes only those rows that meet the criteria specified in the SELECT statement. As a result, each row must have a Department value of Popular, the rows must be grouped together according to the values in the Category column, and the result set must include the number of compact disks in each category; however, as a result of the HAVING clause, only those categories that contain fewer than three compact disks are included.

In this exercise, you create two SELECT statements that each include a HAVING clause. The HAVING clauses are used in conjunction with GROUP BY clauses to further refine the search results.

Try It Out Defining a HAVING Clause in Your SELECT Statement

The following steps describe how to create these 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 SELECT statement restricts the rows returned to only those with a Total value greater than 1. Execute the following SELECT statement at the mysql command prompt:

```
SELECT RatingID, COUNT(*) AS Total
FROM DVDs
GROUP BY RatingID
HAVING Total>1;
```

You should receive results similar to the following:

```
+----------+-------+
| RatingID | Total |
+----------+-------+
| NR       |     4 |
| PG       |     2 |
+----------+-------+
2 rows in set (0.00 sec)
```

3. The next SELECT statement includes a HAVING clause that restricts the rows to only those that have an Amount value greater than 2. Execute the following SELECT statement at the mysql command prompt:

```
SELECT EmpID, COUNT(*) AS Amount
FROM Orders
GROUP BY EmpID
HAVING Amount>2;
```

You should receive results similar to the following:

```
+-------+--------+
| EmpID | Amount |
+-------+--------+
|     2 |      3 |
+-------+--------+
1 row in set (0.09 sec)
```

How It Works

In this exercise, you created two `SELECT` statements, both of which contain `GROUP BY` and `HAVING` clauses. In addition, both statements include the `COUNT(*)` expression in the `SELECT` clause. For example, the first statement that you created determines the number of DVDs for each RatingID, as shown in the following statement:

```
SELECT RatingID, Count(*) AS Total
FROM DVDs
GROUP BY RatingID
HAVING Total>1;
```

As you can see, the result set is grouped together based on the values in the RatingID column. In addition, the total number of rows for each category is provided in the Total column. The result set includes only rows with a Total value greater than 1.

The second `SELECT` statement that you created in this exercise is similar to the first one and contains the same elements. The primary difference is that it groups data together in the Orders table, rather than the DVDs table, and the returned rows must have an Amount value greater than 2. Otherwise, the elements between the two statements are the same.

The ORDER BY Clause

In Chapter 6, you were introduced to the `ORDER BY` clause when you learned about `UPDATE` and `DELETE` statements. As you'll recall from both those statements, you could include an `ORDER BY` clause that allowed you to sort rows that were updated or deleted by one or more columns. The `SELECT` statement also includes an `ORDER BY` clause that allows you to determine the order in which rows are returned in a results set. The following syntax describes the elements in an `ORDER BY` clause:

```
ORDER BY <order by definition>

<order by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
```

As the syntax indicates, the `ORDER BY` clause must include the `ORDER BY` keywords and at least one column name. You can also specify a column alias in place of the actual name. If you include more than one column, a comma must separate them.

For each column that you include in an ORDER BY clause, you can specify whether the rows are sorted in ascending order (the ASC option) or descending order (the DESC option). If neither option is specified, the ASC option is assumed. In addition, when more than one column is specified, the rows are sorted first by the column that is listed first, then by the next specified column, and so on.

Now take a look at an example that uses the ORDER BY clause. The following SELECT statement includes an ORDER BY clause that sorts the rows in the result set according to the values in the CDName column:

```
SELECT CDName, InStock, OnOrder
FROM CDs
WHERE InStock>20
ORDER BY CDName DESC;
```

Notice that the ORDER BY clause specifies that the values should be sorted in descending order, as shown in the following results:

```
+---------------------------+---------+---------+
| CDName                    | InStock | OnOrder |
+---------------------------+---------+---------+
| The Magic of Satie        |      42 |      17 |
| Swan Lake                 |      25 |      44 |
| Stages                    |      42 |       0 |
| Richland Woman Blues      |      22 |       5 |
| Pure                      |      32 |       3 |
| Music for Solo Violin     |      24 |       2 |
| Morimur (after J. S. Bach) |      28 |      17 |
| Golden Road               |      23 |      10 |
| Embrace                   |      24 |      11 |
| Bach Cantatas             |      23 |      12 |
| 25 Classical Favorites    |      32 |      15 |
+---------------------------+---------+---------+
11 rows in set (0.00 sec)
```

As you can see, the result set includes values from the CDName, InStock, and OnOrder columns. Only rows with an InStock value greater than 20 are included here. In addition, the rows that are included are sorted according to the values in the CDName column.

The next example SELECT statement includes an ORDER BY clause that sorts the result set according to two columns:

```
SELECT Department, Category, CDName
FROM CDs
WHERE (InStock+OnOrder-Reserved)<15
ORDER BY Department DESC, Category ASC;
```

In this case, the returned rows are sorted first by the values in the Department column (in descending order) and then by the values in the Category column (in ascending order). The following results show you what you would expect if you executed this statement:

```
+------------+----------+-------------------------------+
| Department | Category | CDName                        |
+------------+----------+-------------------------------+
| Popular    | Country  | Mud on the Tires              |
| Popular    | Rock     | Bloodshot                     |
| Classical  | Dance    | Music for Ballet Class        |
| Classical  | Opera    | The Most Favorite Opera Duets |
| Classical  | Vocal    | Cie li di Toscana             |
+------------+----------+-------------------------------+
5 rows in set (0.00 sec)
```

As you can see, the rows that contain a Department value of Popular are listed first, followed by the Classical rows. In addition, for each group of values, the Category values are sorted in ascending order. For example, the Classical rows are sorted by the Category values Dance, Opera, and Vocal, in that order.

You can sort your result sets by as many columns as it is practical; however, this is useful only if the columns listed first have enough repeated values to make sorting additional columns return meaningful results.

In the following Try It Out you create `SELECT` statements that use the `ORDER BY` clause to sort the values returned by your queries.

Try It Out Defining an ORDER BY Clause in Your SELECT Statement

The following steps describe how to create statements that employ the `ORDER BY` clause:

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 retrieves rows from the DVDs table and orders those rows according to the values in the DVDName column. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT DVDName, MTypeID, RatingID
FROM DVDs
WHERE RatingID!='NR'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+----------------+---------+----------+
| DVDName        | MTypeID | RatingID |
+----------------+---------+----------+
| Amadeus        | mt11    | PG       |
| Mash           | mt12    | R        |
| Out of Africa  | mt11    | PG       |
| What's Up, Doc? | mt12   | G        |
+----------------+---------+----------+
4 rows in set (0.00 sec)
```

3. The next SELECT statement also retrieves rows from the DVDs table, only this time, the rows are ordered by values in the MTypeID column and the RatingID column. Execute the following SELECT statement at the mysql command prompt:

```
SELECT MTypeID, RatingID, DVDName
FROM DVDs
WHERE RatingID!='NR'
ORDER BY MTypeID, RatingID;
```

You should receive results similar to the following:

```
+---------+----------+-----------------+
| MTypeID | RatingID | DVDName         |
+---------+----------+-----------------+
| mt11    | PG       | Out of Africa   |
| mt11    | PG       | Amadeus         |
| mt12    | G        | What's Up, Doc? |
| mt12    | R        | Mash            |
+---------+----------+-----------------+
4 rows in set (0.00 sec)
```

How It Works

The first SELECT statement that you created in this exercise retrieved data from the DVDs table. The data returned by the statement was sorted according to the values in the DVDName column, as the following statement shows:

```
SELECT DVDName, MTypeID, RatingID
FROM DVDs
WHERE RatingID!='NR'
ORDER BY DVDName;
```

The SELECT statement returns values from the DVDName, MTypeID, and RatingID columns. In addition, the statement returns all rows except those that have a RatingID value of NR. The exclamation point/equal sign (!=) combination acts as a comparison operator that means not equal. In other words, the RatingID value cannot equal NR. The SELECT statement goes on to define a sort order by including an ORDER BY clause that specifies the DVDName column. As a result, the rows returned in the result set are sorted according to the values in the DVDName column. In addition, because you specified no ASC or DESC option in the ORDER BY clause, the rows are sorted in ascending order.

The next statement that you created in this exercise included two columns in the ORDER BY clause:

```
SELECT MTypeID, RatingID, DVDName
FROM DVDs
WHERE RatingID!='NR'
ORDER BY MTypeID, RatingID;
```

In this case, the same columns are displayed as in the previous SELECT statement. The columns are specified in a different order in the SELECT clause, however, and the ORDER BY clause includes two columns. As a result, the rows are sorted first according to the values in the MTypeID column, then sorted according to the values in the RatingID column.

The LIMIT Clause

The final clause to review in this chapter is the LIMIT clause. As with the ORDER BY clause, you also saw this clause used in Chapter 6 when you were introduced to the UPDATE and DELETE statements. As was the case with those statements, the LIMIT clause is used most effectively in a SELECT statement when it is used with an ORDER BY clause.

The LIMIT clause takes two arguments, as the following syntax shows:

```
LIMIT [<offset>,] <row count>
```

The first option, <offset>, is optional and indicates where to begin the LIMIT row count. If no value is specified, 0 is assumed. (The first row in a result set is considered to be 0, rather than 1.) The second argument, <row count> in the LIMIT clause, indicates the number of rows to be returned. For example, the following SELECT statement includes a LIMIT clause that specifies a row count of 4.

```
SELECT CDID, CDName, InStock
FROM CDs
WHERE Department='Classical'
ORDER BY CDID DESC
LIMIT 4;
```

As you can see in this statement, no offset value is specified, so 0 is assumed. The row count value is specified as 4, though, so the first four rows of the result set are returned, as shown in the following results:

```
+------+-------------------------------------+---------+
| CDID | CDName                              | InStock |
+------+-------------------------------------+---------+
|   24 | Bach: Six Unaccompanied Cello Suites |      16 |
|   21 | The Best of Italian Opera           |      10 |
|   20 | Morimur (after J. S. Bach)          |      28 |
|   16 | Bach Cantatas                       |      23 |
+------+-------------------------------------+---------+
4 rows in set (0.00 sec)
```

If you were to specify an offset value, the rows returned would begin with the first row indicated by the offset value and end after the number of rows indicated by the row count is returned. For example, the following SELECT statement includes a LIMIT clause that specifies an offset value of 3 and a row count value of 4:

```
SELECT CDID, CDName, InStock
FROM CDs
WHERE Department='Classical'
ORDER BY CDID DESC
LIMIT 3,4;
```

Because the offset value is 3, the result set begins with the fourth row returned by the results. (Remember that the first row is numbered 0.) The result set then includes the four rows that begin with row number 3, as shown in the following results:

```
+------+------------------------+---------+
| CDID | CDName                 | InStock |
+------+------------------------+---------+
|   16 | Bach Cantatas          |      23 |
|   15 | La Boheme              |      20 |
|   14 | 25 Classical Favorites |      32 |
|   13 | Swan Lake              |      25 |
+------+------------------------+---------+
4 rows in set (0.00 sec)
```

As you can see, the result set includes only four rows, numbers 3 through 6. As you might also notice, the rows returned are based on the sort order defined in the `ORDER BY` clause, which specifies that the rows should be sorted by the values in the CDID column, in descending order. In other words, the result set is limited only to the most recent orders that have been added to the table, excluding the first three orders.

In this Try It Out, you create a `SELECT` statement that uses the `LIMIT` clause, in conjunction with the `ORDER BY` clause, to limit the number of rows returned by your query.

Try It Out Defining a LIMIT Clause in Your SELECT Statement

To use the `LIMIT` clause in a `SELECT` statement, 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. Now create a `SELECT` statement that uses an `ORDER BY` clause and a `LIMIT` clause to return the last order added to the Orders table. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT OrderID, CustID
FROM Orders
ORDER BY OrderID DESC
LIMIT 1;
```

You should receive results similar to the following:

```
+---------+--------+
| OrderID | CustID |
+---------+--------+
|      13 |      5 |
+---------+--------+
1 row in set (0.00 sec)
```

How It Works

In this exercise, you created one `SELECT` statement that included an `ORDER BY` clause and a `LIMIT` clause, as shown in the following statement:

```
SELECT OrderID, CustID
FROM Orders
ORDER BY OrderID DESC
LIMIT 1;
```

The statement retrieves values from the OrderID and CustID columns of the Orders table. The returned rows are sorted in descending order according to the values in the OrderID column. The `LIMIT` clause limits the returned rows to only the first row of the results returned by the SELECT statement. As a result, this statement returns only the most recent order added to the Orders table.

Summary

As the chapter has demonstrated, the `SELECT` statement can contain many components, allowing you to create statements as simple or as complex as necessary to retrieve specific data from the tables in your MySQL database. You can specify which columns to retrieve, which rows to retrieve, how the rows should be sorted, whether the rows should be grouped together and summarized, and the number of rows to return in your result set. To provide you with the information you need to create robust `SELECT` statements, the chapter gave you the information necessary to perform the following tasks:

- Create `SELECT` statements that retrieve all columns and all rows from a table
- Create `SELECT` statements that retrieve specific columns from a table
- Assign aliases to column names
- Use expressions in the `SELECT` clauses of your `SELECT` statements
- Use `SELECT` statements to create variables that can be used in later `SELECT` statements
- Create `SELECT` statements that return information that is not based on data in a table
- Add options to your `SELECT` statements
- Add `WHERE` clauses to your `SELECT` statements that determine which rows the statements would return
- Add `GROUP BY` clauses to your `SELECT` statements to generate summary data
- Add `HAVING` clauses to your `SELECT` statements to refine the results returned by summarized data
- Add `ORDER BY` clauses to your `SELECT` statements to sort the rows returned by your statements
- Add `LIMIT` clauses to your `SELECT` statement to limit the number of rows returned by the statement

In later chapters, you learn how to refine your `SELECT` statements even further. For example, you learn how to join tables in a `SELECT` statement or to embed other `SELECT` statements in a parent `SELECT` statement. In addition, you also learn more about using operators and functions to create powerful expressions in your statements. To prepare for these chapters, the information in this chapter provided you with the foundation necessary to perform these more advanced tasks. As a result, you might find it useful to refer to this chapter as you progress through the book.

Exercises

The following exercises are provided as a way for you to better acquaint yourself with the material covered in this chapter. The exercises are based on the example CDs table used for the examples in this chapter. To view the answers to these questions, see Appendix A.

1. You are creating a `SELECT` statement that retrieves data from the CDs table. Your query results should include values from the CDName, InStock, OnOrder, and Reserved columns. In addition, the results should include all rows in the table. What `SELECT` statement should you create?
2. You want to modify the `SELECT` statement that you created in Exercise 1 so that the result set includes an additional column. The column should add the values of the InStock and OnOrder columns and then subtract the value in the Reserved column. You want to assign the Total alias to the new column. What `SELECT` statement should you create?
3. Your next step is to modify the `SELECT` statement that you created in Exercise 2. You plan to limit the rows returned to those rows that have a Department value of Classical and an InStock value less than 20. What `SELECT` statement should you create?
4. You now want to create a `SELECT` statement that summarizes data in the CDs table. The result set should include data that is grouped together by the Department column and then by the Category column. The summary information should include the number of rows for each category, as well as totals for all categories in each department. In addition, the summary column in the result set should be assigned the alias Total, and all grouped columns should be sorted in ascending order. What `SELECT` statement should you create?
5. You now want to modify the `SELECT` statement that you created in Exercise 4 to limit the rows returned to those with a Total value less than 3. What `SELECT` statement should you create?
6. You are creating a `SELECT` statement that retrieves data from the CDs table. Your query results should include values from the CDName column. The values should be sorted in descending order. What `SELECT` statement should you create?