8

Using Operators in Your SQL Statements

In previous chapters, you have seen a number of expressions used within SQL statements to help define the actions taken by those statements. For example, you can use expressions in the `WHERE` clauses of `SELECT`, `UPDATE`, and `DELETE` statements to help identify which rows in a table or tables should be acted upon. An expression, as you'll recall, is a formula made up of column names, literal values, operators, and functions. Together, these components allow you to create expressions that refine your SQL statements to effectively query and modify data within your MySQL database.

In order to allow the components of an expression to work effectively with each other, operators are used to define those interactions and to specify conditions that limit the range of values permitted in a result set. An *operator* is a symbol or keyword that specifies a specific action or condition between other elements of an expression or between expressions. For example, the addition (+) operator specifies that two elements within an expression should be added together. In this chapter, you learn how to create expressions that use the various operators supported by MySQL. Specifically, this chapter covers the following topics:

- Which components make up MySQL expressions, how operators allow expression elements to interact with each other, how operators are prioritized within an expression, and how to use parentheses to group components of an expression together.
- Which categories of operators MySQL supports, including arithmetic, comparison, logical, bit, and sort operators, and how to use those operators in expressions contained within SQL statements.

Creating MySQL Expressions

As you have seen throughout the book, expressions can play in important role in any of your SQL data manipulation statements, including `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements. Although the complexity of the expression can vary greatly from statement to statement, the basic elements that can make up an expression are the same:

- **Column names:** When a column name is listed in an expression, it refers to the value contained within the column for the specific row that the SQL statement affects. For example, suppose the InStock column for a specific row contains a value of 14. If the expression references the InStock column, it is replaced with the value 14.
- **Literal values:** This refers to a value that is used in an expression exactly as entered into that expression. Also referred to as a *literal* or *constant*, a literal value can be a string, number, or date/time value. For example, if an expression contains the value 14, the expression uses that value exactly as written.
- **Functions:** A function performs a predefined task and returns some type of result, such as a numerical, string, or date/time value. When using a function, you must often supply arguments that provide the function with the information it needs to perform its task. These arguments are usually in the form of column names, literal values, or other expressions.
- **Operators:** An operator is used in conjunction with column names, literal values, and functions to calculate and compare values within an expression. MySQL supports arithmetic, comparison, logical, bit, and sort operators, which are discussed later in the chapter.

Most expressions consist of at least one argument and one operator, although an expression can consist of nothing more than a function. However, for a typical expression, several elements are included. The term *argument* is generally used to describe the non-operator part of an expression, such as the column name or the literal value. For example, suppose you have an SQL statement that includes the following WHERE clause:

```
WHERE InStock>14
```

The expression in this clause is `InStock>14`. The expression includes two arguments — InStock and 14 — and the greater than (>) comparison operator. When an expression is used in a `WHERE` clause (or in a `HAVING` clause in a `SELECT` statement), the expression is often referred to as a *condition*. This is because each expression is evaluated to determine whether the condition is true, false, or `NULL`. If a `WHERE` clause contains multiple expressions, you can say that it has multiple conditions, each of which must be evaluated.

As you progress through this chapter, you see how to use column names, literal values, and operators to create expressions. You even see a couple examples of functions. However, most of the discussion about functions is held off until Chapter 9, where you learn how to incorporate functions into your expressions and see how they work in conjunction with column names, literal values, and operators to create comprehensive, precise expressions in your SQL statements.

Operator Precedence

When an expression in an SQL statement is processed, it is evaluated according to the order in which elements are included in the statement and the precedence in which operators are assigned. MySQL processes expressions according to a very specific operator precedence. The following list shows the operator precedence used when processing expressions in an SQL statement:

1. `BINARY`, `COLLATE`
2. `NOT` (logical negation), `!` (logical negation)
3. - (unary minus), ~ (unary bit inversion)

4. ^ (bitwise exclusive OR comparison)
5. * (multiplication), / (division), % (modulo)
6. - (subtraction), + (addition)
7. << (bitwise shift left), >> (bitwise shift right)
8. & (bitwise AND)
9. | (bitwise OR)
10. All comparison operators except BETWEEN and NOT BETWEEN
11. BETWEEN, NOT BETWEEN
12. AND (logical addition), && (logical addition)
13. OR (logical OR comparison), || (logical OR comparison), XOR (logical exclusive OR comparison)

The operators listed here are shown from the highest precedence to the lowest. For example, the BINARY operator has precedence over the BETWEEN and ampersand (&) operators. However, operators that appear on the same line of the list have the same level of precedence, so they are evaluated in the order in which they appear in the expression. For example, the multiplication (*) and division (/) operators have the same level of precedence so they are evaluated in the order in which they appear.

> *You learn the function of each of these operators as your progress through the chapter. As you start using operators, refer to the preceding list as necessary to understand how a particular operator is prioritized within the order of precedence.*

Grouping Operators

Because of operator precedence, you may often find that, in order to control how expressions and group of expressions are evaluated, you need to group together the appropriate elements within parentheses in order to ensure that those elements are processed as a unit. For example, because the multiplication (*) operator has precedence over the addition (+) operator, the following expression is evaluated as follows:

```
3+4*5=23
```

In this expression, 4 and 5 are multiplied, and *then* 3 is added to the sum. However, you can group together the arguments in an expression to better control your results, as shown in the following example:

```
(3+4)*5=35
```

In this case, the 3+4 calculation is treated as a unit. As a result, 7 and 5 are multiplied, resulting in a total of 35, rather than the 23 in the previous expression. As you learn more about the operators that MySQL supports and how they're used within an expression, you'll get a better sense of how to group elements together. However, as a general rule, it's a good idea to use parentheses whenever any possibility of confusion exists to make certain that your expressions are readable and to ensure that they are correct.

Using Operators in Expressions

MySQL supports a number of different types of operators, which can be divided into the following five categories:

- **Arithmetic operators:** Perform calculations on the arguments within an expression.
- **Comparison operators:** Compare the arguments in an expression to test whether a condition is true, false, or `NULL`.
- **Logical operators:** Verify the validity of one or more expressions to test whether they return a condition of true, false, or `NULL`.
- **Bitwise operators:** Manipulate the bit values associated with numerical values.
- **Sort operators:** Specify the collation and case-sensitivity of searches and sorting operations.

The rest of the chapter focuses on the operators supported in each of these categories.

Arithmetic Operators

Arithmetic operators are used to calculate arguments within an expression. They are similar to the symbols found in algebraic equations in that they are used to add, subtract, multiply, and divide values. The following table provides a brief description of the arithmetic operators supported by MySQL.

Operator	Description
+ (addition)	Adds the two arguments together.
- (subtraction)	Subtracts the second argument from the first argument.
- (unary)	Changes the sign of the argument.
* (multiplication)	Multiplies the two arguments together.
/ (division)	Divides the first argument by the second argument.
% (modulo)	Divides the first argument by the second argument and provides the remainder from that operation.

In earlier chapters, you saw several examples of arithmetic operators used within SQL statements. Now you take a more thorough look at these types of operators by examining statements that use several of them. The examples in this section are based on a table named Inventory, which is shown in the following table definition:

```
CREATE TABLE Inventory
(
   ProductID SMALLINT NOT NULL PRIMARY KEY,
   InStock SMALLINT NOT NULL,
   OnOrder SMALLINT NOT NULL,
   Reserved SMALLINT NOT NULL
);
```

For the purposes of the examples in this section, you can assume that the Inventory table is populated with the values shown in the following INSERT statement:

```
INSERT INTO Inventory
VALUES (101, 10, 15, 4), (102, 16, 9, 3), (103, 15, 2, 13);
```

As you can see, three rows have been added to the Inventory table. Now suppose that you want to add another row. However, you want the OnOrder value to be based on the InStock value, as shown in the following example:

```
INSERT INTO Inventory
VALUES (104, 16, 25-InStock, 0);
```

As you can see, the third value in the VALUES clause includes the expression 25-InStock. The expression contains two arguments — 25 and InStock — and it contains the subtraction (-) operator. As a result, the value from the InStock column, which in this case is 16, is subtracted from 25, giving you a total value of 9. This means that 9 is inserted into the OnOrder column of the Inventory table when the row is added to the table.

Although referencing a column in an expression in the VALUES clause of an INSERT statement can be a handy approach, you can reference a column only if it has already been assigned a value. For example, you would not want to reference the OnOrder column to assign a value to the InStock column because no value has yet been assigned to that column. Be sure that, whenever you plan to reference a column in a VALUES clause, the referenced column is current and contains the correct value.

Now take a look at a SELECT statement that contains an expression that uses arithmetic operators. In the following statement, the second element in the select list includes an expression:

```
SELECT ProductID, InStock+OnOrder-Reserved AS Available
FROM Inventory;
```

The select list in this statement includes two elements. The first is the column name ProductID, and the second is the expression InStock+OnOrder-Reserved. The expression includes three arguments — InStock, OnOrder, and Reserved — and two arithmetic operators — the addition (+) operator and the subtraction (-) operator. Because both operators share the same level of precedence, the values in the InStock and OnOrder columns are added together first, and then, from that sum, the value in the Reserved column is subtracted. The following result set shows how a value has been calculated in the Available column for each row:

```
+-----------+-----------+
| ProductID | Available |
+-----------+-----------+
|       101 |        21 |
|       102 |        22 |
|       103 |         4 |
|       104 |        25 |
+-----------+-----------+
4 rows in set (0.01 sec)
```

The select list in a SELECT statement can also include more than one expression, as the following statement demonstrates:

```
SELECT ProductID, InStock+OnOrder-Reserved AS Available,
   InStock+OnOrder*2-Reserved AS DoubleOrder
FROM Inventory;
```

This statement is similar to the previous example except that the result set returned by the modified statement includes a third column named DoubleOrder. The new column is based on an expression that is identical to the first expression except that it doubles the value in the OnOrder column. The result of this is a value that shows how many items would be available if you doubled the number on order. To achieve this, the expression uses the multiplication (*) arithmetic operator to multiply the OnOrder value by two. That amount is then added to the InStock value, and the Reserved value is then subtracted from the total. Because the multiplication operator takes precedence over the subtraction and addition operators, the multiplication operation is carried out first.

Take a look at the first row to help demonstrate how this works. For the row with a ProductID value of 101, the InStock value is 10, the OnOrder value is 15, and the Reserved value is 4. Based on the expression, you must first multiply the OnOrder value by 2, which is 15 x 2, or 30. You then add 30 to the InStock value of 10, to give you 40, and then subtract the Reserved value of 4, which leaves you a total of 36, as the following result set shows:

```
+-----------+-----------+-------------+
| ProductID | Available | DoubleOrder |
+-----------+-----------+-------------+
|       101 |        21 |          36 |
|       102 |        22 |          31 |
|       103 |         4 |           6 |
|       104 |        25 |          34 |
+-----------+-----------+-------------+
4 rows in set (0.00 sec)
```

As you can see, the DoubleOrder column contains the modified totals. The intent of the statement is to calculate the number of available items if the number on order were doubled. However, suppose you want to double the entire amount of available items and you tried to create a statement similar to the following:

```
SELECT ProductID, InStock+OnOrder-Reserved*2 AS Doubled
FROM Inventory;
```

Because of operator precedence, the Reserved value is first doubled, the InStock value is added to the OnOrder value, and then the doubled Reserved value is subtracted from that total. For example, for the row with the ProductID value of 101, the Reserved value is 4, which means that it is doubled to 8. The InStock value of 10 is then added to the OnOrder value of 15, which gives you a total a 25. The double reserved value of 8 is then subtracted from the 25, giving you a final total of 17, as shown in the following result set:

```
+-----------+---------+
| ProductID | Doubled |
+-----------+---------+
|       101 |      17 |
|       102 |      19 |
|       103 |      -9 |
|       104 |      25 |
+-----------+---------+
4 rows in set (0.00 sec)
```

These results are fine if your intent is merely to double the number of items that are on reserve; however, if your intent is to determine how many items you would have if you doubled your availability, you would have to modify your statement as follows:

```
SELECT ProductID, (InStock+OnOrder-Reserved)*2 AS Doubled
FROM Inventory;
```

Notice that part of the expression is now enclosed in parentheses, which means that these arguments and operators are processed as a unit. Only then is the amount multiplied by 2, which provides you with an amount that is double what your availability is, as shown in the following results:

```
+-----------+---------+
| ProductID | Doubled |
+-----------+---------+
|       101 |      42 |
|       102 |      44 |
|       103 |       8 |
|       104 |      50 |
+-----------+---------+
4 rows in set (0.00 sec)
```

You are not limited to `INSERT` and `SELECT` statements to use expressions that contain arithmetic operators. For example, the following `UPDATE` statement contains an expression in the `SET` clause:

```
UPDATE Inventory
SET OnOrder=OnOrder/2;
```

In this case, the expression (`OnOrder/2`) uses the division (`/`) operator to divide the value in the OnOrder column by two. Suppose that, after executing the `UPDATE` statement, you ran the following `SELECT` statement:

```
SELECT * FROM Inventory;
```

From this `SELECT` statement, you would receive results similar to the following:

```
+-----------+---------+---------+----------+
| ProductID | InStock | OnOrder | Reserved |
+-----------+---------+---------+----------+
|       101 |      10 |       7 |        4 |
|       102 |      16 |       4 |        3 |
|       103 |      15 |       1 |       13 |
|       104 |      16 |       4 |        0 |
+-----------+---------+---------+----------+
4 rows in set (0.00 sec)
```

As you can see, the OnOrder values have all been divided by two. However, because this is an integer type column, only whole numbers are used, so the values are rounded off for any of the odd numbers that were divided. For example, the OnOrder value for the row that contains a ProductID value of 101 was rounded off to 7.

> *In the example above, the OnOrder value has been rounded down from 7.5 (the value returned by 15/2) to 7. Different implementations of the C library might round off numbers in different ways. For example, some might always round numbers up or always down, while others might always round toward zero.*

In the following exercise, you create several SQL statements that include expressions that use arithmetic operators. The expressions are used in the select lists of `SELECT` statements and the `SET` clause and `WHERE` clauses of an `UPDATE` statement. For this exercise, you use the DVDs table in the DVDRentals database.

Try It Out Creating Expressions with Arithmetic Operators

The following steps describe how to create SQL statements that use arithmetic operators:

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. Your first `SELECT` statement includes an expression in the select list. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, YearRlsd, (YEAR(CURDATE())-YearRlsd) AS YearsAvailable
FROM DVDs;
```

   You should receive results similar to the following:

```
+------------------------------+----------+----------------+
| DVDName                      | YearRlsd | YearsAvailable |
+------------------------------+----------+----------------+
| White Christmas              |     2000 |              4 |
| What's Up, Doc?              |     2001 |              3 |
| Out of Africa                |     2000 |              4 |
| The Maltese Falcon           |     2000 |              4 |
| Amadeus                      |     1997 |              7 |
| The Rocky Horror Picture Show |     2000 |              4 |
| A Room with a View           |     2000 |              4 |
| Mash                         |     2001 |              3 |
+------------------------------+----------+----------------+
8 rows in set (0.24 sec)
```

3. In the next SQL statement, you create an `INSERT` statement that adds a row to the DVDs table. (You use this row to perform an update in the next step.) Execute the following `INSERT` statement at the mysql command prompt:

```
INSERT INTO DVDs
SET DVDName='The Wizard of Oz', NumDisks=2, YearRlsd=1999,
   MTypeID='mt14', StudID='s102', RatingID='G', FormID='f2', StatID='s2';
```

   You should receive a response indicating that the query executed properly, affecting one row.

4. Now you create an `UPDATE` statement that includes an expression in the `SET` clause and in the `WHERE` clause. Execute the following `UPDATE` statement at the mysql command prompt:

```
UPDATE DVDs
SET NumDisks=NumDisks/2
WHERE DVDName='The Wizard of Oz';
```

   You should receive a response indicating that the query executed properly, affecting one row.

5. To return to the DVDs table to its original state, delete the row that you created in Step 3. Execute the following `DELETE` statement at the mysql command prompt:

```
DELETE FROM DVDs
WHERE DVDName='The Wizard of Oz';
```

You should receive a response indicating that the query executed properly, affecting one row.

How It Works

In this exercise, you created two statements that included expressions that contained arithmetic operators. The first of these was the following `SELECT` statement, which includes an expression as an element in the select list:

```
SELECT DVDName, YearRlsd, (YEAR(CURDATE())-YearRlsd) AS YearsAvailable
FROM DVDs;
```

As the statement indicates, the select list first includes the DVDName column and the YearRlsd column. These two column names are then followed by an expression (`YEAR(CURDATE())-YearRlsd`) that subtracts the year value in the YearRlsd column from the current year. The current year is derived by using two functions: the `YEAR()` function and the `CURDATE()` function, which is embedded as an argument in the `YEAR()` function. (Functions are discussed in detail in Chapter 9.) By using these two functions together in this way, you can arrive at the current year. As a result, for each row returned by the `SELECT` statement, the YearRlsd value is subtracted from the current year and placed in a column named YearsAvailable, which is the alias assigned to the expression. The YearRlsd value is subtracted from the current year by using the subtraction (-) arithmetic operator.

The next statement that includes expressions is the `UPDATE` statement, which contains an expression in the `SET` clause and in the `WHERE` clause, as shown in the following statement:

```
UPDATE DVDs
SET NumDisks=NumDisks/2
WHERE DVDName='The Wizard of Oz';
```

The first expression in this statement is in the `SET` clause and appears after the first equal sign: `NumDisks/2`. This expression uses the division (/) operator to divide the value in the NumDisks column by 2. This value is then inserted into the NumDisks column, as indicated by the SET clause. An expression is also used in the `WHERE` clause to limit which rows are updated. The expression (`DVDName='The Wizard of Oz'`) specifies a condition that must be met in order for the row to be updated. In this case, the equals (=) comparison operator specifies that the DVDName value must equal *The Wizard of Oz*. Comparison operators are discussed in the following section.

Comparison Operators

Comparison operators are used to compare the arguments on either side of the expression and determine whether the condition is true, false, or `NULL`. If either argument is `NULL` or if both arguments are `NULL`, the condition is considered `NULL`. The only exception to this is the `NULL`-safe (`<=>`) operator, which evaluates to true when both arguments are the same, even if they are both `NULL`. For a condition to be acceptable, it must evaluate to true. For example, suppose you have a `SELECT` statement that includes the following `WHERE` clause:

```
SELECT ProductName, ProductType
FROM Products
WHERE ProductType='Boat';
```

The `WHERE` clause includes the expression `ProductType='Boat'`. When the table is queried, the ProductType value in each row is compared to the value Boat. If the value equals Boat, the condition is true. If the value does not equal Boat, the condition is false. If the ProductType value is `NULL`, the condition is `NULL`. As a result, only rows that contain a ProductType value of Boat meet the condition. In other words, the condition evaluates to true for those rows, and those are the rows returned in the results set.

MySQL supports a number of comparison operators that allow you to define various types of conditions in your SQL statements. The following table describes each of these operators.

Operator	Description
`=`	Evaluates to true if both arguments are equal, unless both conditions are `NULL`.
`<=>`	Evaluates to true if both arguments are equal, even if both conditions are `NULL`.
`<>, !=`	Evaluates to true if the two arguments are not equal.
`<`	Evaluates to true if the value of the first argument is less than the value of the second argument.
`<=`	Evaluates to true if the value of the first argument is less than or equal to the value of the second argument.
`>`	Evaluates to true if the value of the first argument is greater than the value of the second argument.
`>=`	Evaluates to true if the value of the first argument is greater than or equal to the value of the second argument.
`IS NULL`	Evaluates to true if the argument equals a null value.
`IS NOT NULL`	Evaluates to true if the argument does not equal a null value.
`BETWEEN`	Evaluates to true if the value of the argument falls within the range specified by the `BETWEEN` clause.
`NOT BETWEEN`	Evaluates to true if the value of the argument does not fall within the range specified by the `NOT BETWEEN` clause.
`IN`	Evaluates to true if the value of the argument is specified within the `IN` clause.
`NOT IN`	Evaluates to true if the argument is not specified within the `NOT IN` clause.
`LIKE`	Evaluates to true if the value of the argument is not specified by the `LIKE` construction.
`NOT LIKE`	Evaluates to true if the value of the argument is not specified by the `NOT LIKE` construction.

Operator	Description
REGEXP	Evaluates to true if the value of the argument is specified by the REGEXP construction.
NOT REGEXP	Evaluates to true if the value of the argument is not specified by the NOT REGEXP construction.

As you can see, there are many comparison operators, and the best way to better understand them is to look at example statements that use these operators. The examples in this section are based on 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)
);
```

For the purposes of the examples in this section, you can assume that the following INSERT statement was used to populate 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', NULL),
('Cie li di Toscana', 16, 6, 8, 'Classical', NULL),
('Mississippi Blues', 2, 25, 6, 'Popular', 'Blues'),
('Pure', 32, 3, 10, 'Popular', NULL),
('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', NULL),
('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', NULL),
('Stages', 42, 0, 8, 'Popular', 'Blues'),
('Bach: Six Unaccompanied Cello Suites', 16, 8, 8, 'Classical', 'General');
```

The first example that you review is a SELECT statement whose WHERE clause contains an expression that uses a comparison operator:

```
SELECT CDName, Department, Category
FROM CDs
WHERE CDID=3;
```

As you can see, an equals (=) comparison operator specifies that the CDID value in each row returned by the query must contain a value equal to 3. The WHERE clause expression includes two arguments — CDID and 3 — along with the equals operator. Because the table includes only one row that contains a CDID value of 3, that is the only row for which the WHERE clause condition evaluates to true. As a result, no other row is retuned, as shown in the following result set:

```
+------------------+------------+----------+
| CDName           | Department | Category |
+------------------+------------+----------+
| New Orleans Jazz | Popular    | Jazz     |
+------------------+------------+----------+
1 row in set (0.00 sec)
```

The next example SELECT statement contains a WHERE clause whose expression specifies that the Category column must contain NULL:

```
SELECT CDName, Department, Category
FROM CDs
WHERE Category=NULL
ORDER BY CDName;
```

As you can see, the expression includes two arguments — Category and NULL — and the equals operator. If you return to the original INSERT statement that added data to the CDs table, you see that five rows contain a Category value of NULL. However, if you execute this statement, you should receive the following results:

```
Empty set (0.00 sec)
```

In this case, no rows are returned because, when using an equals (=) comparison operator, neither side is permitted to equal NULL in order for the condition to evaluate to true. As a result, MySQL interprets the condition as NULL, so no rows are returned, even though some rows do indeed contain NULL. One way to get around this is to use the NULL-safe (<=>) comparison operator, as shown in the following statement:

```
SELECT CDName, Department, Category
FROM CDs
WHERE Category<=>NULL
ORDER BY CDName;
```

As you can see, the only difference between this statement and the previous statement is the use of the NULL-safe operator. Now when you execute the statement, you receive the following results:

```
+----------------------+------------+----------+
| CDName               | Department | Category |
+----------------------+------------+----------+
| Cie li di Toscana    | Classical  | NULL     |
| Music for Solo Violin | Classical  | NULL     |
| Pure                 | Popular    | NULL     |
| Runaway Soul         | Popular    | NULL     |
| The Magic of Satie   | Classical  | NULL     |
+----------------------+------------+----------+
5 rows in set (0.00 sec)
```

Another way to work with columns that contain NULL is to use the IS NULL comparison operator, as shown in the following statement:

```
SELECT CDName, Department, Category
FROM CDs
WHERE Category IS NULL
ORDER BY CDName;
```

In this case, the expression has been changed to include only the column name and the IS NULL keywords. When you execute this statement, you should receive the same results as the preceding SELECT statement.

Several of the comparison operators allow you to use the NOT keyword to reverse the meaning of the operator. For example, if you were to rewrite the preceding SELECT statement to include the IS NOT NULL operator, your statement would be as follows:

```
SELECT CDName, Department, Category
FROM CDs
WHERE Category IS NOT NULL
ORDER BY CDName;
```

The only difference between this statement and the preceding one is the addition of the keyword NOT. However, as the following result set shows, the addition of this one element is indeed significant:

```
+-------------------------------------+------------+----------+
| CDName                              | Department | Category |
+-------------------------------------+------------+----------+
| 25 Classical Favorites              | Classical  | General  |
| Bach Cantatas                       | Classical  | General  |
| Bach: Six Unaccompanied Cello Suites | Classical  | General  |
| Bloodshot                           | Popular    | Rock     |
| Embrace                             | Popular    | New Age  |
| Golden Road                         | Popular    | Country  |
| La Boheme                           | Classical  | Opera    |
| Live in Paris                       | Popular    | Jazz     |
| Mississippi Blues                   | Popular    | Blues    |
| Morimur (after J. S. Bach)          | Classical  | General  |
| Mud on the Tires                    | Popular    | Country  |
| Music for Ballet Class              | Classical  | Dance    |
| New Orleans Jazz                    | Popular    | Jazz     |
| Richland Woman Blues                | Popular    | Blues    |
| Stages                              | Popular    | Blues    |
| Swan Lake                           | Classical  | Dance    |
| The Best of Italian Opera           | Classical  | Opera    |
| The Essence                         | Popular    | New Age  |
| The Most Favorite Opera Duets       | Classical  | Opera    |
+-------------------------------------+------------+----------+
19 rows in set (0.00 sec)
```

By adding the NOT keyword, the results now include those rows that do *not* contain a Category value of NULL, rather than those that do contain NULL. No rows that contain a Category value of NULL are included in this result set.

Now take a look at another type of comparison operator. In the following example, the WHERE clause includes an expression that compares a calculated value to a value of 20:

```
SELECT CDName, InStock, OnOrder, Reserved
FROM CDs
WHERE (InStock+OnOrder-Reserved)>20
ORDER BY CDName;
```

As you can see, this expression uses arithmetic operators and a comparison operator. The arithmetic operators are used to derive a value from the InStock, OnOrder, and Reserved columns. The operators and columns are enclosed in parentheses to group together the arguments to ensure that the correct value is derived from these columns. The resulting value is then compared to the value of 20. Because the greater than (>) comparison operator is used, the value derived from the three columns must be greater than 20. As a result, only those rows for which this condition evaluates to true are returned by SELECT statement, as shown in the following result set:

```
+---------------------------+---------+---------+----------+
| CDName                    | InStock | OnOrder | Reserved |
+---------------------------+---------+---------+----------+
| 25 Classical Favorites    |      32 |      15 |       12 |
| Bach Cantatas             |      23 |      12 |        8 |
| Embrace                   |      24 |      11 |       14 |
| La Boheme                 |      20 |      10 |        5 |
| Live in Paris             |      18 |      20 |       10 |
| Mississippi Blues         |       2 |      25 |        6 |
| Morimur (after J. S. Bach)|      28 |      17 |       16 |
| Music for Solo Violin     |      24 |       2 |        5 |
| Pure                      |      32 |       3 |       10 |
| Runaway Soul              |      15 |      30 |       14 |
| Stages                    |      42 |       0 |        8 |
| Swan Lake                 |      25 |      44 |       28 |
| The Best of Italian Opera |      10 |      35 |       12 |
| The Magic of Satie        |      42 |      17 |       17 |
+---------------------------+---------+---------+----------+
14 rows in set (0.02 sec)
```

You can verify that the correct data was returned by sampling one of the rows. For example, the first row contains an InStock value of 32, an OnOrder value of 15, and a Reserved value of 12. According to the expression in the WHERE clause, this would read (32+15-12)>20, or 35>20, which is a true condition.

As you can see, each comparison operator has a very specific meaning. For example, suppose you change the last example simply by changing the comparison operator, as shown in the following SELECT statement:

```
SELECT CDName, InStock, OnOrder, Reserved
FROM CDs
WHERE (InStock+OnOrder-Reserved)<20
ORDER BY CDName;
```

Now the value returned by the InStock, OnOrder, and Reserved columns must equal an amount less than 20. As a result, you would receive the following result set:

```
+-------------------------------------+---------+---------+----------+
| CDName                              | InStock | OnOrder | Reserved |
+-------------------------------------+---------+---------+----------+
| Bach: Six Unaccompanied Cello Suites |      16 |       8 |        8 |
| Bloodshot                           |      10 |       5 |        3 |
| Cie li di Toscana                   |      16 |       6 |        8 |
| Golden Road                         |      23 |      10 |       17 |
| Mud on the Tires                    |      12 |      15 |       13 |
| Music for Ballet Class              |       9 |       4 |        2 |
| The Essence                         |       5 |      20 |       10 |
| The Most Favorite Opera Duets       |      10 |       5 |        3 |
+-------------------------------------+---------+---------+----------+
8 rows in set (0.00 sec)
```

Another comparison operator that can be very useful is the `IN` operator. In the following example, the expression specifies that the Category value must be Blues or Jazz:

```
SELECT CDName, Category, InStock
FROM CDs
WHERE Category IN ('Blues', 'Jazz')
ORDER BY CDName;
```

The `WHERE` clause expression — `Category IN ('Blues', 'Jazz')` — contains several elements. It first specifies the Categories column, then the `IN` keyword, and then a list of values, which are enclosed in parentheses and separated by a comma. For this `SELECT` statement to return a row, the Category value must equal Blues or Jazz. The condition evaluates to true only for these rows. Rows that contain other Category values, including `NULL`, are not returned, as shown in the following result set:

```
+---------------------+----------+---------+
| CDName              | Category | InStock |
+---------------------+----------+---------+
| Live in Paris       | Jazz     |      18 |
| Mississippi Blues   | Blues    |       2 |
| New Orleans Jazz    | Jazz     |      17 |
| Richland Woman Blues | Blues    |      22 |
| Stages              | Blues    |      42 |
+---------------------+----------+---------+
5 rows in set (0.00 sec)
```

MySQL also allows you to use operators to specify a range of values. The following example uses the `NOT BETWEEN` operator to specify which rows should not be included in the result set:

```
SELECT CDName, InStock, OnOrder, Reserved
FROM CDs
WHERE (InStock+OnOrder-Reserved) NOT BETWEEN 10 AND 20
ORDER BY CDName;
```

The expression used in this case — `(InStock+OnOrder-Reserved) NOT BETWEEN 10 AND 20` — first uses arithmetic operators to derive a value from the InStock, OnOrder, and Reserved columns. The `NOT`

BETWEEN operator (along with the AND keyword) defines the range in which values cannot be included. In other words, the value derived from the InStock, OnOrder, and Reserved columns cannot fall within the range of 10 through 20, inclusive, as shown in the follow result set:

```
+---------------------------+---------+---------+----------+
| CDName                    | InStock | OnOrder | Reserved |
+---------------------------+---------+---------+----------+
| 25 Classical Favorites    |      32 |      15 |       12 |
| Bach Cantatas             |      23 |      12 |        8 |
| Embrace                   |      24 |      11 |       14 |
| La Boheme                 |      20 |      10 |        5 |
| Live in Paris             |      18 |      20 |       10 |
| Mississippi Blues         |       2 |      25 |        6 |
| Morimur (after J. S. Bach)|      28 |      17 |       16 |
| Music for Solo Violin     |      24 |       2 |        5 |
| Pure                      |      32 |       3 |       10 |
| Runaway Soul              |      15 |      30 |       14 |
| Stages                    |      42 |       0 |        8 |
| Swan Lake                 |      25 |      44 |       28 |
| The Best of Italian Opera |      10 |      35 |       12 |
| The Magic of Satie        |      42 |      17 |       17 |
+---------------------------+---------+---------+----------+
14 rows in set (0.00 sec)
```

If you were to take a row and calculate the InStock, OnOrder, and Reserved columns (according to the how they're calculated in the expression), your total would either be less than 10 or greater than 20, but rows whose totals fall within that range would be returned.

Another useful comparison operator is the LIKE operator, which allows you to search for values similar to a specified value. The LIKE operator supports the use of two wildcards:

- **Percentage (%):** Represents zero or more values.
- **Underscore (_):** Represents exactly one value.

The following SELECT statement includes a WHERE clause expression that searches for CDName values that contain "bach" somewhere in its title:

```
SELECT CDName, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE CDName LIKE '%bach%'
ORDER BY CDName;
```

Because the WHERE clause uses the percentage wildcard both before and after the value bach, any characters can fall before and after that value, as shown in the following result set:

```
+-------------------------------------+-----------+
| CDName                              | Available |
+-------------------------------------+-----------+
| Bach Cantatas                       |        27 |
| Bach: Six Unaccompanied Cello Suites |        16 |
| Morimur (after J. S. Bach)          |        29 |
+-------------------------------------+-----------+
3 rows in set (0.01 sec)
```

MySQL supports yet another operator that allows you to locate values similar to a specified value. The REGEXP comparison operator allows you to specify a number of different options and configurations in order to return similar values. The following table lists the primary options that you can use with the REGEXP operator to create expressions in your SQL statements.

Options	Meaning	Example	Acceptable Values
<value>	The tested value must contain the specified value.	'bo'	about, book, abbot, boot
<^>	The tested value must *not* contain the specified value.	'^bo'	abut, took, amount, root
.	The tested value can contain any individual character represented by the period (.).	'b.'	by, be, big, abbey
[<characters>]	The tested value must contain at least one of the characters listed within the brackets.	'[xz]'	dizzy, zebra, x-ray, extra
[<range>]	The tested value must contain at least one of the characters listed within the range of values enclosed by the brackets.	'[1-5]'	15, 3, 346, 50, 22, 791
^	The tested value must begin with the value preceded by the caret (^) symbol.	'^b'	book, big, banana, bike
$	The tested value must end with the value followed by the dollar sign ($) symbol.	'st$'	test, resist, persist
*	The tested value must include zero or more of the character that precedes the asterisk (*).	'^b.*e$'	bake, be, bare, battle

The REGEXP operator can seem confusing at first until you see a couple statements that show how it's used. For example, the following SELECT statement uses the REGEXP operator to return rows that contain a CDName value that begins with the letters a through f:

```
SELECT CDName, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE CDName REGEXP '^[a-f]'
ORDER BY CDName;
```

The expression — `CDName REGEXP '^[a-f]'` — first specifies the CDName column, the `REGEXP` keyword, and the value to be matched. The value is enclosed in single quotes and contains the caret (^) symbol and a bracketed range that specifies the letters a through f. Because the caret is used, the next specified value must appear at the beginning of the column values. In this case, the specified value is actually a bracketed range. This means that, for the condition to evaluate to true for a particular row, the CDName value must begin with the letter a through f, as shown in the following result set:

```
+-------------------------------------+-----------+
| CDName                              | Available |
+-------------------------------------+-----------+
| Bach Cantatas                       |        27 |
| Bach: Six Unaccompanied Cello Suites |        16 |
| Bloodshot                           |        12 |
| Cie li di Toscana                   |        14 |
| Embrace                             |        21 |
+-------------------------------------+-----------+
5 rows in set (0.00 sec)
```

You can even be more specific with the `REGEXP` operator by extending the specified value used by the operator, as shown in the following `SELECT` statement:

```
SELECT CDName, InStock
FROM CDs
WHERE CDName REGEXP '^[mn].*[sz]$'
ORDER BY CDName;
```

In this statement, the `REGEXP` value again begins with a caret, indicating that the next value must appear at the beginning of the column value. However, in this case, a range of values is not specified in the brackets, but rather two specific characters: m and n. As a result, the CDName value must begin with an m or an n. In addition, the `REGEXP` value includes the period/asterisk (`.*`) construction. The period (.) indicates that any single character can be included, and the asterisk (*) indicates that the preceding character can be repeated zero or more times. In another words, any character can be repeated any number of times. The `REGEXP` value then ends with the bracketed s and z, followed by a dollar ($) sign. As a result, the returned CDName value must end in an s or a z, as shown in the follow result set:

```
+------------------------+---------+
| CDName                 | InStock |
+------------------------+---------+
| Mississippi Blues      |       2 |
| Mud on the Tires       |      12 |
| Music for Ballet Class |       9 |
| New Orleans Jazz       |      17 |
+------------------------+---------+
4 rows in set (0.00 sec)
```

As you can see in the results, each CDName value begins with an m or an n and ends with an s or a z, and any characters can be included between the beginning and ending letters. These four rows were the only rows that met the condition specified by the `WHERE` clause. In other words, these were the only rows for which the `WHERE` clause expression evaluated to true.

As you saw in this section, comparison operators provide a great deal of flexibility in allowing you to create SQL statements that are both flexible and very specific. The following exercise has you create a number of `SELECT` statements that use comparison operators in the `WHERE` clause to define which rows your query returns. You query tables in the DVDRentals database.

Try It Out Creating Expressions with Comparison Operators

The following steps describe how to create `SELECT` statements employing comparison operators in the `WHERE` 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 requests information from the DVDs table. Your statement returns only those rows that have a StatID value of s2 (Available). Execute the following `SELECT` statement at the mysql command prompt:

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

   You should receive results similar to the following:

```
+------------------------------+--------+
| DVDName                      | StatID |
+------------------------------+--------+
| Amadeus                      | s2     |
| Mash                         | s2     |
| The Maltese Falcon           | s2     |
| The Rocky Horror Picture Show | s2     |
| What's Up, Doc?              | s2     |
+------------------------------+--------+
5 rows in set (0.01 sec)
```

3. The next `SELECT` statement that you create is similar to the last, except that you will return only those rows that do *not* have a StatID value of s2. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT DVDName, StatID
FROM DVDs
WHERE StatID<>'s2'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+--------------------+--------+
| DVDName            | StatID |
+--------------------+--------+
| A Room with a View | s1     |
| Out of Africa      | s1     |
| White Christmas    | s1     |
+--------------------+--------+
3 rows in set (0.00 sec)
```

4. The next `SELECT` statement that you create retrieves those rows for DVDs that have been released after the year 2000. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT DVDName, YearRlsd, StatID
FROM DVDs
WHERE YearRlsd>2000
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+-----------------+----------+--------+
| DVDName         | YearRlsd | StatID |
+-----------------+----------+--------+
| Mash            |     2001 | s2     |
| What's Up, Doc? |     2001 | s2     |
+-----------------+----------+--------+
2 rows in set (0.01 sec)
```

5. The next `SELECT` statement returns results from the Employees table. Only those rows that do not contain an EmpMN value of `NULL` should be included in the result set. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT EmpFN, EmpMN, EmpLN
FROM Employees
WHERE EmpMN IS NOT NULL;
```

You should receive results similar to the following:

```
+-------+-------+----------+
| EmpFN | EmpMN | EmpLN    |
+-------+-------+----------+
| John  | P.    | Smith    |
| Mary  | Marie | Michaels |
| Rita  | C.    | Carter   |
+-------+-------+----------+
3 rows in set (0.00 sec)
```

6. Now you query data in the Transactions table. Your `SELECT` statement should return only these rows that have a DVDID of 2, 5, or 8. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT OrderID, TransID, DVDID
FROM Transactions
WHERE DVDID IN (2, 5, 8)
ORDER BY OrderID, TransID;
```

You should receive results similar to the following:

```
+---------+---------+-------+
| OrderID | TransID | DVDID |
+---------+---------+-------+
|       1 |       3 |     8 |
|       6 |      10 |     2 |
|       8 |      13 |     2 |
|      10 |      18 |     5 |
|      11 |      20 |     2 |
|      11 |      21 |     8 |
|      12 |      22 |     5 |
+---------+---------+-------+
7 rows in set (0.01 sec)
```

7. In the next `SELECT` statement, you query the DVDs table and return any rows that contain the word "horror" anywhere within the DVDName value. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName LIKE '%horror%'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+------------------------------+--------+----------+
| DVDName                      | StatID | RatingID |
+------------------------------+--------+----------+
| The Rocky Horror Picture Show | s2     | NR       |
+------------------------------+--------+----------+
1 row in set (0.01 sec)
```

8. The final `SELECT` statement that you create in this exercise returns any row with a DVDName value that contains the letters "ro" anywhere within the name. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName REGEXP 'ro'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+------------------------------+--------+----------+
| DVDName                      | StatID | RatingID |
+------------------------------+--------+----------+
| A Room with a View           | s1     | NR       |
| The Rocky Horror Picture Show | s2     | NR       |
+------------------------------+--------+----------+
2 rows in set (0.00 sec)
```

How It Works

In this exercise, you created seven SELECT statements that retrieved data from tables in the DVDRentals database. The first statement included a WHERE clause expression that specified the value of the StatID column.

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

For this statement to return a row, the StatID value must equal s2. For each row that contains a StatID value of s2, the condition specified in the WHERE clause expression evaluates to true. All other rows evaluate to false. No rows evaluate to NULL because null values are not permitted in this column.

The next SELECT statement that you created was identical to the first except that you used a not equal (<>) operator to specify that the StatID value should not equal s2, as shown in the following statement:

```
SELECT DVDName, StatID
FROM DVDs
WHERE StatID<>'s2'
ORDER BY DVDName;
```

As a result of this statement, the only rows that meet the WHERE clause condition were those that contain a non-NULL value other than s2. All other rows evaluate to false.

Your next SELECT statement used the greater than (>) comparison operator to compare the YearRlsd values in the DVDs table to the year 2000, as shown in the following statement:

```
SELECT DVDName, YearRlsd, StatID
FROM DVDs
WHERE YearRlsd>2000
ORDER BY DVDName;
```

The only rows that return a true condition are those for DVDs that were released after the year 2000. All other rows fail to meet the condition specified by the WHERE expression.

You then created the following SELECT statement, which uses the IS NOT NULL operator to determine which rows to return:

```
SELECT EmpFN, EmpMN, EmpLN
FROM Employees
WHERE EmpMN IS NOT NULL;
```

The expression in this statement specifies the EmpMN column and the IS NOT NULL operator. As a result, only rows that contain an EmpMN value other than NULL are included in the result set. In other words, only employees who listed a middle name are included in the result set.

Next you created a SELECT statement that retrieved data from the Transactions table. The statement retrieved rows with DVDID values of 2, 5, or 8, as shown in the following statement:

```
SELECT OrderID, TransID, DVDID
FROM Transactions
WHERE DVDID IN (2, 5, 8)
ORDER BY OrderID, TransID;
```

The expression in this case includes the name of the DVDID column, the IN keyword, and three values that are enclosed in parentheses and separated by commas. In order for a condition to be true, a row must have a DVDID value that is equal to one of the values specified as an argument for the IN operator. As a result, only rows with a DVDID value of 2, 5, or 8 are returned.

Next you created a SELECT statement that used the LIKE operator to return rows, as shown in the following statement:

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName LIKE '%horror%'
ORDER BY DVDName;
```

The WHERE clause expression shown here specifies the DVDName column, the LIKE keyword, and a value enclosed in single quotes. The value uses percentage (%) wildcards to indicate that any characters can appear before or after the word horror. As a result, only rows with a DVDName value that contains the word horror are included in the result set, which in this case is only one row.

The final SELECT statement that you created also used an operator to define a value to be used to match patterns, as shown in the following statement

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName REGEXP 'ro'
ORDER BY DVDName;
```

In this statement, the REGEXP operator searches for any values in the DVDName column that contains the letters “ro,” in that order. However, the letters can appear anywhere within the DVDName value. In this case, only two rows are returned because only two DVDName values return a WHERE clause condition of true.

Logical Operators

Logical operators allow you to test the validity of one or more expressions. Through the use of these operators, you can associate expressions to determine whether the conditions, when taken as a whole, evaluate to true, false, or NULL. For a condition or set of conditions to be acceptable, they must evaluate to true. The following table describes the logical operators available in MySQL.

Operator	Description
AND	Evaluates to true if both of the two arguments or expressions evaluate to true. You can use double ampersands (&&) in place of the AND operator.
OR	Evaluates to true if either of the two arguments or expressions evaluates to true. You can use the double vertical pipes (||) in place of the OR operator
XOR	Evaluates to true if exactly one of the two arguments or expressions evaluates to true.
NOT, !	Evaluates to true if the argument or expression evaluates to false. You can use an exclamation point (!) in place of the NOT operator.

To better understand how to use logical operators, take a look at a few examples. These examples are based on the following table definition:

```
CREATE TABLE Books
(
   BookID SMALLINT NOT NULL PRIMARY KEY,
   BookName VARCHAR(40) NOT NULL,
   Category VARCHAR(15),
   InStock SMALLINT NOT NULL,
   OnOrder SMALLINT NOT NULL
);
```

For the purposes of these examples, you can assume that the following INSERT statement has been used to insert data into the Books table:

```
INSERT INTO Books
VALUES (101, 'Noncomformity: Writing on Writing', 'Nonfiction', 12, 13),
(102, 'The Shipping News', 'Fiction', 17, 20),
(103, 'Hell\'s Angels', 'Nonfiction', 23, 33),
(104, 'Letters to a Young Poet', 'Nonfiction', 32, 12),
(105, 'A Confederacy of Dunces', 'Fiction', 6, 35),
(106, 'One Hundred Years of Solitude', 'Fiction', 28, 14),
(107, 'Where I\'m Calling From', NULL, 46, 3);
```

The first example is a SELECT statement that includes a WHERE clause that contains two expressions (conditions):

```
SELECT BookName, Category, InStock, OnOrder
FROM Books
WHERE Category='Fiction' AND (InStock+OnOrder)>40
ORDER BY BookName;
```

The first expression in the WHERE clause specifies that the Category column must contain the value Fiction. The second expression specifies that the sum derived from the InStock and OnOrder columns must be greater than 40. The two expressions are connected by the AND logical operator. As a result, both expressions must evaluate to true in order for the conditions, when taken as a whole, to evaluate to true. In other words, each row returned by the SELECT statement must contain a Category value of Fiction *and* an (InStock+OnOrder) value greater than 40, as shown in the following result set:

```
+------------------------------+----------+---------+---------+
| BookName                     | Category | InStock | OnOrder |
+------------------------------+----------+---------+---------+
| A Confederacy of Dunces      | Fiction  |       6 |      35 |
| One Hundred Years of Solitude | Fiction  |      28 |      14 |
+------------------------------+----------+---------+---------+
2 rows in set (0.00 sec)
```

The next `SELECT` statement also includes two expressions within the `WHERE` clause, only this time the two expressions are connected by an `OR` logical operator, as shown in the following statement:

```
SELECT BookName, Category, InStock, OnOrder
FROM Books
WHERE InStock>30 OR OnOrder>30
ORDER BY BookName;
```

The first expression specifies that the InStock column must contain a value greater than 30, and the second expression specifies that the OnOrder value must be greater than 30. Because an `OR` operator connects these two conditions, only one of the expressions must evaluate to true. In other words, the InStock value must be greater than 30 *or* the OnOrder value must be greater than 30, as the following result set demonstrates:

```
+-------------------------+------------+---------+---------+
| BookName                | Category   | InStock | OnOrder |
+-------------------------+------------+---------+---------+
| A Confederacy of Dunces | Fiction    |       6 |      35 |
| Hell's Angels           | Nonfiction |      23 |      33 |
| Letters to a Young Poet | Nonfiction |      32 |      12 |
| Where I'm Calling From  | NULL       |      46 |       3 |
+-------------------------+------------+---------+---------+
4 rows in set (0.00 sec)
```

You can also use an `XOR` logical operator between two expressions in an SQL statement, as the following example shows:

```
SELECT BookName, Category, InStock, OnOrder
FROM Books
WHERE Category='Fiction' XOR InStock IS NULL
ORDER BY BookName;
```

When you use an `XOR` operator to compare expressions, the expressions, when taken as a whole, evaluate to true if exactly one of the expressions evaluates to true. In other words, one expression can evaluate to true, but not both. As a result, either the Category column must contain a value of Fiction *or* the InStock column must contain a value of `NULL`. However, both conditions cannot be true, as shown in the following result set:

```
+------------------------------+----------+---------+---------+
| BookName                     | Category | InStock | OnOrder |
+------------------------------+----------+---------+---------+
| A Confederacy of Dunces      | Fiction  |       6 |      35 |
| One Hundred Years of Solitude | Fiction  |      28 |      14 |
| The Shipping News            | Fiction  |      17 |      20 |
+------------------------------+----------+---------+---------+
3 rows in set (0.01 sec)
```

The following SELECT statement includes three expressions connected with an AND operator and an OR operator:

```
SELECT BookName, Category, InStock, OnOrder
FROM Books
WHERE InStock>20 AND (Category IS NULL OR NOT (Category='Fiction'))
ORDER BY BookName;
```

The first expression in this statement specifies that the InStock column must contain a value greater than 20. The next two expressions, which are connected by an OR operator, are enclosed in parentheses, so they're evaluated together. The first of these conditions specifies that the Category column must contain a NULL value. The NOT operator precedes the second of these two conditions, which means that the Category column must *not* contain the value Fiction. Because these two expressions are connected by an OR operator, either condition can evaluate to true. As a result, for a row to be returned, the InStock value must be greater than 20 *and* the Category column must contain a value of NULL or a value other than Fiction, as shown in the following result set:

```
+-------------------------+------------+---------+---------+
| BookName                | Category   | InStock | OnOrder |
+-------------------------+------------+---------+---------+
| Hell's Angels           | Nonfiction |      23 |      33 |
| Letters to a Young Poet | Nonfiction |      32 |      12 |
| Where I'm Calling From  | NULL       |      46 |       3 |
+-------------------------+------------+---------+---------+
3 rows in set (0.00 sec)
```

As the example SELECT statements demonstrate, you can use logical operators to create complex statements that allow you to include multiple expressions in your statements in order to specify the exact rows that you want to return. You can also use logical operators the WHERE clauses of your UPDATE and DELETE statements to specify which rows should be modified in your MySQL tables.

In the following exercise, you create several SELECT statements that include expressions that contain logical operators that create conditions made up of multiple expressions. In the statements, you query data from the DVDs table in the DVDRentals database.

Try It Out — Creating Expressions with Logical Operators

The following steps describe how to create the statements containing logical operators:

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 two WHERE clause expressions that are linked together with an OR logical operator. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDName, MTypeID, RatingID
FROM DVDs
WHERE RatingID='G' OR RatingID='PG'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+-----------------+---------+----------+
| DVDName         | MTypeID | RatingID |
+-----------------+---------+----------+
| Amadeus         | mt11    | PG       |
| Out of Africa   | mt11    | PG       |
| What's Up, Doc? | mt12    | G        |
+-----------------+---------+----------+
3 rows in set (0.04 sec)
```

3. Next, create a `SELECT` statement that includes three expressions in the `WHERE` clause. The clause includes an `AND` operator and an `OR` operator. Execute the following `SELECT` statement at the mysql command prompt:

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

You should receive results similar to the following:

```
+-----------------+---------+----------+
| DVDName         | MTypeID | RatingID |
+-----------------+---------+----------+
| Amadeus         | mt11    | PG       |
| What's Up, Doc? | mt12    | G        |
+-----------------+---------+----------+
2 rows in set (0.02 sec)
```

4. The next `SELECT` statement also includes three expressions connected by an `AND` operator and an `OR` operator. In addition, use the `NOT` operator to reverse one of sets of conditions. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT DVDName, MTypeID, RatingID
FROM DVDs
WHERE StatID='s2' AND NOT (RatingID='G' OR RatingID='PG')
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+------------------------------+---------+----------+
| DVDName                      | MTypeID | RatingID |
+------------------------------+---------+----------+
| Mash                         | mt12    | R        |
| The Maltese Falcon           | mt11    | NR       |
| The Rocky Horror Picture Show | mt12    | NR       |
+------------------------------+---------+----------+
3 rows in set (0.01 sec)
```

How It Works

The first SELECT statement includes two expressions in the WHERE clause, as shown in the following statement:

```
SELECT DVDName, MTypeID, RatingID
FROM DVDs
WHERE RatingID='G' OR RatingID='PG'
ORDER BY DVDName;
```

The first expression specifies that the RatingID value must be G, and the second expression specifies that the RatingID value must be PG. Because you used an OR logical operator to connect the two expressions, either expression can evaluate to true in order for a row to be included in the result set.

The next SELECT statement that you created included three expressions:

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

The first expression specifies that the StatID value must be s2, the second expression specifies that the RatingID value must be G, and the third expression specifies that the RatingID value must be PG. However, the last two expressions are enclosed in parentheses and they are connected by an OR logical operator, so only one of these two conditions must evaluate to true. However, because these conditions are connected to the first condition by an AND operator, the first condition must also evaluate to true. In other words, the StatID value must be s2 and the RatingID value must be G or PG.

The last SELECT statement that you created in this exercise is nearly identical to the previous one, except that you added the NOT logical operator before the parentheses that enclose the last to expressions:

```
SELECT DVDName, MTypeID, RatingID
FROM DVDs
WHERE StatID='s2' AND NOT (RatingID='G' OR RatingID='PG')
ORDER BY DVDName;
```

Because you included the NOT operator, the condition specified within the parentheses (the last two expressions) is negated, so the opposite condition must be met. As a result, the StatID value must be s2, and the RatingID value *cannot* be G or PG.

Bitwise Operators

Bitwise operators are a special type of operator that allow you to compare and modify the bit values associated with numerical values stored in your database. The following table lists the bitwise operators available in MySQL.

Operator	Description
&	The bitwise AND operator that compares bits and returns 1 when each bit equals 1. Otherwise, 0 is returned.
|	The bitwise OR operator that compares bits and returns 1 when at least one of the bits equals 1. Otherwise, 0 is returned.
^	The bitwise XOR operator that compares bits and returns 1 if exactly one of the bits equals 1. Otherwise, 0 is returned.
~	The bitwise negation operator that inverts all bits in a specified number. All 0 bits are converted to 1, and all 1 bits are converted to 0.
<<	The bitwise shift left operator that shifts all bits to the left by the specified number of positions.
>>	The bitwise shift right operator that shifts all bits to the right by the specified number of positions.

You can use the bitwise operators to work directly with numerical values stored within a table. For example, suppose that your database includes a table that stores the attribute settings for specific users of an application. The following table definition provides an example of this type of table:

```
CREATE TABLE Attributes
(
   UserID SMALLINT NOT NULL PRIMARY KEY,
   Settings TINYINT UNSIGNED NOT NULL
);
```

Each row within the table stores the attributes for a specific user. The the UserID column, which is the primary key, indicates the user, and the Settings column stores the application attributes for each user. The Settings column is configured with an unsigned TINYINT data type, so it stores one byte of data, which ranges from 0 through 255. Now suppose that you use the following INSERT statement to add the attribute settings for three users:

```
INSERT INTO Attributes
VALUES (101, 58), (102, 73), (103, 45);
```

For each set of values added to the table, the first value is the UserID value and the second value is the Settings value. Because bitwise operators are concerned with working with bit values, each Settings value is associated with one byte, as shown in Figure 8-1. For example, user 101 contains a Settings value of 58. The byte that represents this number is 00111010 (bit 32 + bit 16+ bit 8 + bit 2 = 58). This means that for user 101, bits 32, 16, 8, and 2 have been set to 1, and all other bits within the byte have been set to 0. Figure 8-1 also shows the bit settings for users 102 and 103.

UserID	Settings
101	58
102	73
103	45

128	64	32	16	8	4	2	1
0	0	1	1	1	0	1	0
0	1	0	0	1	0	0	1
0	0	1	0	1	1	0	1

Figure 8-1

Now suppose that you want to update the settings for user 101 so that bit 1 is also set to 1, as is the case for users 102 and 103. To update row 101, you can use the following `UPDATE` statement:

```
UPDATE Attributes
SET Settings=Settings | 1
WHERE UserID=101;
```

The `UPDATE` statement specifies that the bit value in the Settings column (for the row with a UserID value of 101) should be compared to the value of 1 and updated appropriately. Because you use the bitwise `OR` (|) operator, each bit within the byte is compared. Any bit position that includes a bit value of 1 returns a value of 1. Any bit position that contains only bit values of 0 returns a 0. Figure 8-2 demonstrates how this works. The byte associated with the value 58 is 00111010. The byte that is associated with the value 1 is 00000001. The last row in Figure 8-2 shows the results of the comparisons between the two bytes. For example, bit 16 is set to 1 for value 58 and set to 0 for value 1, so the bitwise `OR` operator returns a value of 1 for bit 16. As a result, the new value inserted into the Settings column is 59 (bit 32 + bit 16+ bit 8+ bit 2 + bit 1 = 59).

Settings	128	64	32	16	8	4	2	1
58	0	0	1	1	1	0	1	0
1	0	0	0	0	0	0	0	1
59	0	0	1	1	1	0	1	1

Figure 8-2

You can confirm that the correct change has been made to the Attributes table by running the following `SELECT` statement:

```
SELECT * FROM Attributes;
```

When you execute the statement, you should receive the following result set:

```
+--------+----------+
| UserID | Settings |
+--------+----------+
|    101 |       59 |
|    102 |       73 |
|    103 |       45 |
+--------+----------+
3 rows in set (0.00 sec)
```

As you can see, the Settings value for the row with a UserID value of 101 is now set to 59.

As you saw earlier in this section, you can also use bitwise operators to manipulate bits in other ways. For example, the following `UPDATE` statement moves each bit to the left one position:

```
UPDATE Attributes
SET Settings=Settings << 1;
```

The bitwise shift left (<<) operator indicate that the bits should be shifted to the left. The value to the right of the carets indicates how many positions the bits should be moved. Figure 8-3 demonstrates what the bits look like after they've been moved to the left one position. Notice that the Settings values have now been modified to reflect the new bit settings. For example, the value for user 101 is now 118 (bit 64 + bit 32 + bit 16+ bit 4+ bit 2 = 118).

UserID	Settings
101	118
102	146
103	90

128	64	32	16	8	4	2	1
0	1	1	1	0	1	1	0
1	0	0	1	0	0	1	0
0	1	0	1	1	0	1	0

Figure 8-3

You can verify the new settings by running the following `SELECT` statement:

```
SELECT * FROM Attributes;
```

The following result set shows the new values that have been inserted into the Settings column of the Attributes table:

```
+--------+----------+
| UserID | Settings |
+--------+----------+
|    101 |      118 |
|    102 |      146 |
|    103 |       90 |
+--------+----------+
3 rows in set (0.00 sec)
```

As you can see, each Settings value has been updated as a result of using the bitwise operator in your `UPDATE` statement.

> *The examples that you have seen in this section are based on only eight bits (one byte) of data. However, bitwise operators support calculations up to 64 buts. As a result, you can perform a bitwise shift left operation as long as there are bits to the left, but you cannot go beyond the 64th bit. For example, if you were to repeat the* `UPDATE` *statement shown in the previous example, the bits for user 102 would be shifted out of the first byte into the byte to the left. This would place a bit in bit position 9 (bit 1 of the second byte). As a result, your calculation would begin with bit 1 in the second byte and move to the right accordingly.*

In the following Try It Out exercise, you create several `SELECT` statements that demonstrate how you can use bitwise operators to convert numerical values that represent bit values. The `SELECT` statements that you create are made up only of `SELECT` clauses, without specifying any tables or other clauses. The purpose of this exercise is only to demonstrate how the bitwise operators work.

Try It Out Creating Expressions with Bitwise Operators

To create these statements using bitwise operators, follow these steps:

1. Open the mysql client utility.
2. In the first `SELECT` statement, you use the bitwise `AND` (`&`) operator to manipulate the bit values. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT 8 & 8, 8 & 10, 8 & 16;
```

You should receive results similar to the following:

```
+-------+--------+--------+
| 8 & 8 | 8 & 10 | 8 & 16 |
+-------+--------+--------+
|     8 |      8 |      0 |
+-------+--------+--------+
1 row in set (0.03 sec)
```

3. In the next `SELECT` statement, you use the bitwise `OR` (`|`) operator to manipulate the bit values. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT 8 | 8, 8 | 10, 8 | 16;
```

You should receive results similar to the following:

```
+-------+--------+--------+
| 8 | 8 | 8 | 10 | 8 | 16 |
+-------+--------+--------+
|     8 |     10 |     24 |
+-------+--------+--------+
1 row in set (0.00 sec)
```

4. The final `SELECT` statement that you create in this exercise uses the bitwise `XOR` (`^`) operator to manipulate the bit values. Execute the following `SELECT` statement at the mysql command prompt:

```
SELECT 8 ^ 8, 8 ^ 10, 8 ^ 16;
```

You should receive results similar to the following:

```
+-------+--------+--------+
| 8 ^ 8 | 8 ^ 10 | 8 ^ 16 |
+-------+--------+--------+
|     0 |      2 |     24 |
+-------+--------+--------+
1 row in set (0.02 sec)
```

How It Works

For this exercise, you created several `SELECT` statements that used bitwise operators to calculate numerical values based on manipulating the underlying bit values. The first `SELECT` statement used the bitwise `AND` operator, as shown in the following statement:

```
SELECT 8 & 8, 8 & 10, 8 & 16;
```

The SELECT statement is made up only of a SELECT clause that includes three expressions. Each expression contains two arguments, which are numerical values and the bitwise AND operator. The bitwise AND operator compares each bit in the two values and returns 1 if both compared bits are 1 and returns 0 if either of the compared bits or both of those bits is 0. Figure 8-4 demonstrates how bits are compared for the values in each expression.

Value	128	64	32	16	8	4	2	1
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
10	0	0	0	0	1	0	1	0
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
16	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0

Figure 8-4

For example, the second expression uses the bitwise AND operator to compare a value of 8 to a value of 10. The byte that represents the value 8 includes a 1 in the bit 8 position and a 0 in each of the other positions. The byte that represents the value 10 has a 1 in the bit 8 position, a 1 in the bit 2 position, and a 0 in each of the other bit positions. Because both bit 8 positions contain 1, a 1 is returned. Because only one bit 2 position contains a 1, a 0 is returned. However, because all other bit positions are 0, a 0 is returned for those positions. As a result, the value produced by the comparison contains a 1 only in the bit 8 position and nowhere else, which means that the expression produces a value of 8.

The next SELECT statement that you created in this exercise is nearly identical to the last statement, except that you used a bitwise OR operator, rather than an ampersand, as shown in the following statement.

```
SELECT 8 | 8, 8 | 10, 8 | 16;
```

As you can see, this statement also contains three expressions. However, when the bits are compared in each expression, a 1 is returned if one or both bits contain a value of 1, as shown in Figure 8-5. For example, the second expression compares a 10 to 8. Because both values contain a 1 in the bit 8 position, a 1 is returned. In addition, because the second value contains a value of 1 in the bit 2 position, a 1 is returned for that position as well. All other bit positions return a 0 because the compared bits each contain a 0. As a result, the value produced by this expression contains a 1 in the bit 8 position and a 1 in the bit 2 position, which results in a numerical value of 10.

The final SELECT statement that you created is also like the previous statements, except that it uses the bitwise XOR operator, as shown in the following SELECT statement:

```
SELECT 8 ^ 8, 8 ^ 10, 8 ^ 16;
```

Value	128	64	32	16	8	4	2	1
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
10	0	0	0	0	1	0	1	0
10	0	0	0	0	1	0	1	0
8	0	0	0	0	1	0	0	0
16	0	0	0	1	0	0	0	0
24	0	0	0	1	1	0	0	0

Figure 8-5

In this case, when the bits are compared, a value of 1 is returned only if one of the two bits contains a value of 1, as shown in Figure 8-6. As you can see, if both bits contain a 1, a 0 value is returned. If both bits contain a 0 value, a 0 is again returned. However, if one bit contains a 1 and the other contains a 0, a 1 is returned. For example, in the second expression, bit 2 contains a 0 for the first value and a 1 for the second value, so a 1 is returned. However, bit 8 contains a 1 for both values, so a 0 is returned. In addition, all other bit positions contain a 0 for each value, so a 0 is returned for each of these bit positions. As a result, the value produced by the expression contains a 1 only in bit 2, so the numerical value returned by that expression is 2.

Value	128	64	32	16	8	4	2	1
8	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0
10	0	0	0	0	1	0	1	0
2	0	0	0	0	0	0	1	0
8	0	0	0	0	1	0	0	0
16	0	0	0	1	0	0	0	0
24	0	0	0	1	1	0	0	0

Figure 8-6

Sort Operators

The final type of operators covered in this chapter are the sort operators, which are used to define a pattern that is compared to values within a column. The rows returned are based on whether the compared values match the specified pattern. The following table describes each of the sort operators.

Operator	Description
BINARY	Converts a string to a binary string so that comparing and sorting data is case-sensitive.
COLLATE	Specifies that a particular collation be used to compare and sort string data.

The best way to understand each of these operators is to look at examples. The examples are based on the following table definition:

```
CREATE TABLE ProductColors
(
   ProdID SMALLINT NOT NULL PRIMARY KEY,
   ProdColor VARCHAR(15) NOT NULL
);
```

For these examples, you can assume that the following INSERT statement was used to populate the ProductColors table:

```
INSERT INTO ProductColors
VALUES (101, 'Red'), (102, 'red'), (103, 'RED'), (104, 'REd'), (105, 'reD'),
   (106, 'Blue'), (107, 'blue'), (108, 'BLUE'), (109, 'BLue'), (110, 'blUE');
```

Notice that the values added to the table include only the primary key values for the first column and some form of the values red or blue for the second columns. The values are added in this way merely to demonstrate how sort operators work.

The first sort operator discussed is the BINARY operator. However, before you see an example of a statement that includes a BINARY operator, take a look at the following SELECT statement:

```
SELECT * FROM ProductColors
WHERE ProdColor='red';
```

As you can see, the statement is a basic SELECT statement that retrieves rows from the ProductColors table. The rows returned are only those that contain a ProdColor value of red, as shown in the following results:

```
+--------+-----------+
| ProdID | ProdColor |
+--------+-----------+
|    101 | Red       |
|    102 | red       |
|    103 | RED       |
|    104 | REd       |
|    105 | reD       |
+--------+-----------+
5 rows in set (0.00 sec)
```

As the result set demonstrates, all variations of the red value are returned, regardless of the capitalization used in each value. This is because MySQL, when retrieving data, ignores the case of that data. However, you can override this default behavior by adding the `BINARY` operator to your expression, as shown in the following `SELECT` statement:

```
SELECT * FROM ProductColors
WHERE ProdColor = BINARY 'red';
```

As you can see, this statement is nearly identical to the preceding example, except for the addition of the keyword `BINARY` directly before the value. As a result, the ProdColor column must match the case of the specified value, as well as matching the value itself. As a result, this statement only returns one row, as shown in the following result set:

```
+--------+-----------+
| ProdID | ProdColor |
+--------+-----------+
|    102 | red       |
+--------+-----------+
1 row in set (0.01 sec)
```

As you can see, the value listed in the ProdColor column of the result set is an exact match to the value specified in the `WHERE` clause expression.

The next sort operator supported by MySQL is the `COLLATE` operator, which allows you ro specify a collation in your expression. For example, the following `SELECT` statement specifies that the `latin1_german2_ci` collation be used when determining which rows to retrieve:

```
SELECT * FROM ProductColors
WHERE ProdColor COLLATE latin1_german2_ci = 'red';
```

As you can see, the `WHERE` clause expression in this statement includes the `COLLATE` keyword and the name of the collation (`latin1_german2_ci`). By specifying the collation, the comparison operator (the equals operator in this case) compares values based on the specified collation. If you execute this statement, you receive the following results:

```
+--------+-----------+
| ProdID | ProdColor |
+--------+-----------+
|    101 | Red       |
|    102 | red       |
|    103 | RED       |
|    104 | REd       |
|    105 | reD       |
+--------+-----------+
5 rows in set (0.00 sec)
```

As you may have noticed, the results returned are the same as the results returned when you didn't specify the collation. This is because the specified collation and the default collation treat these particular values the same way when sorting and comparing values. Whenever you specify a collation, you should be well aware of how that collation differs from the default that is used; otherwise you might end up with results you were not looking for. In addition, any collation that you do specify must be supported by the character set being used.

This next exercise allows you to try out the BINARY sort operator in a SELECT statement that retrieves data from the DVDs table in the DVDRentals database. You actually create two SELECT statements in this exercise, one that does not use the BINARY operator and one that does.

Try It Out Creating Expressions with Sort Operators

To create these statements that employ sort operators, 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 that you create does not use the BINARY operator. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName REGEXP 'W'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+------------------------------+--------+----------+
| DVDName                      | StatID | RatingID |
+------------------------------+--------+----------+
| A Room with a View           | s1     | NR       |
| The Rocky Horror Picture Show | s2     | NR       |
| What's Up, Doc?              | s2     | G        |
| White Christmas              | s1     | NR       |
+------------------------------+--------+----------+
4 rows in set (0.00 sec)
```

3. Now create a SELECT statement nearly identical to the one that you created in Step 2, except that the new statement includes the BINARY operator. Execute the following SELECT statement at the mysql command prompt:

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName REGEXP BINARY 'W'
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+-----------------+--------+----------+
| DVDName         | StatID | RatingID |
+-----------------+--------+----------+
| What's Up, Doc? | s2     | G        |
| White Christmas | s1     | NR       |
+-----------------+--------+----------+
2 rows in set (0.05 sec)
```

How It Works

In this statement, you created two SELECT statements. The statements were nearly identical except that the first one did not contain the BINARY operator and the second one did, as shown in the following statement:

```
SELECT DVDName, StatID, RatingID
FROM DVDs
WHERE DVDName REGEXP BINARY 'W'
ORDER BY DVDName;
```

As you can see, the WHERE clause includes an expression that uses the REGEXP operator to compare the letter W to the values in the DVDName clause. The first SELECT statement returned any row that contains a DVDName value that included a W. However, adding the BINARY operator ensured that only those rows that contain an uppercase W were returned.

Summary

As you have seen in this chapter, operators are critical to your ability to create effective expressions, and expressions are essential to creating flexible, robust SQL statements. Operators allow you to perform calculations on the values derived from individual arguments as well as allow you to compare those values. You can also use operators to join the values derived from individual expressions in order to specify a unified condition, and you can specify comparison and sorting criteria in your expressions. Specifically, this chapter provided you the background information and examples necessary to perform the following tasks:

- Use arithmetic operators to perform calculations on the elements within an expression.
- Use comparison operators to compare arguments within an expression in order to test values to determine whether they return a result of true, false, or NULL.
- Use logical operators to join multiple expressions to test whether the expressions, when taken as a whole, return a result of true, false, or NULL.
- Use bitwise operators to compare the actual bit value associated with a numerical value in order to manipulate those bits.
- Use sort operators to specify the collation and case-sensitivity of searching and sorting operations.

Each type of operation can play a critical role in creating effective expressions. However, operators are not the only components that can play a significant part in an expression. Functions provide powerful tools for creating expressions that, when used in conjunction with operators, allow you to manipulate column and literal values to create dynamic data management statements that are precise, flexible, and very effective. For that reason, the next chapter introduces you to the functions that MySQL supports and explains how you can incorporate them into your SQL statements.

Exercises

For these exercises, you create a number of SQL statements that use various types of operators to return results or update rows. The exercises are based on the following table definition:

```
CREATE TABLE Produce
(
   ProdID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ProdName VARCHAR(40) NOT NULL,
   Variety VARCHAR(40) NULL,
   InStock SMALLINT UNSIGNED NOT NULL,
   OnOrder SMALLINT UNSIGNED NOT NULL,
   SeasonAttr TINYINT UNSIGNED NOT NULL
);
```

You can assume that the following `INSERT` statement has been used to populate the Produce table:

```
INSERT INTO Produce
VALUES (101, 'Apples', 'Red Delicious', 2000, 1000, 4),
(102, 'Apples', 'Fuji', 1500, 1200, 4),
(103, 'Apples', 'Golden Delicious', 500, 1000, 4),
(104, 'Apples', 'Granny Smith', 300, 800, 4),
(105, 'Oranges', 'Valencia', 1200, 1600, 15),
(106, 'Oranges', 'Seville', 1300, 1000, 15),
(107, 'Grapes', 'Red seedless', 3500, 1500, 4),
(108, 'Grapes', 'Green seedless', 3500, 1500, 4),
(109, 'Carrots', NULL, 4500, 1500, 6),
(110, 'Broccoli', NULL, 800, 2500, 6),
(111, 'Cherries', 'Bing', 2500, 2500, 2),
(112, 'Cherries', 'Rainier', 1500, 1500, 2),
(113, 'Zucchini', NULL, 1000, 1300, 2),
(114, 'Mushrooms', 'Shitake', 800, 900, 15),
(115, 'Mushrooms', 'Porcini', 400, 600, 15),
(116, 'Mushrooms', 'Portobello', 900, 1100, 15),
(117, 'Cucumbers', NULL, 2500, 1200, 2);
```

Use the Produce table to complete the following exercises. You can find the answers to these exercises in Appendix A.

1. Create a `SELECT` statement that retrieves data from the ProdName, InStock, and OnOrder columns. In addition, the result set should include a column named Total that contains the values of the InStock column added to the OnOrder column. The result set should also be ordered according to the values in the ProdName column.
2. Create a `SELECT` statement that retrieves data from the ProdName, Variety, InStock, and OnOrder columns. The result set should include only rows whose InStock plus OnOrder values are greater than or equal to 5000. The result set should also be ordered according to the values in the ProdName column.
3. Create a `SELECT` statement that retrieves data from the ProdName, Variety, and InStock columns. The rows returned should have an InStock value greater than or equal to 1000. In addition, the rows should contain a ProdName value of Apples or Oranges. The result set should also be ordered according to the values in the ProdName column.
4. Create an `UPDATE` statement that modifies the rows that have a ProdName value of Grapes. The SeasonAttr values within those rows should be modified so that the bit 2 position is set to one, without affecting any of the other bit positions.