9

Using Functions in Your SQL Statements

In earlier chapters, you learned how to use expressions in your SQL statements to make those statements more robust and specific. As you recall, one of the elements that you can use in an expression is a function. Each function performs a specific task and then returns a value that represents the output resulting from the performance of that task. For many functions, you must provide one or more arguments that supply the parameters used by the functions to perform the necessary tasks. These tasks can include calculating numeric data, manipulating string data, returning system data, converting and extracting data, and performing numerous other operations.

In this chapter, you learn about many of the functions included in MySQL. The chapter explains the purpose of each of these functions, describes the results you can expect when a statement includes a function, and provides numerous examples that demonstrate how to use each function. Although this chapter doesn't describe every function included with MySQL, it covers many of them, focusing on those that you're most likely to use when creating SQL statements. Specifically, the chapter covers the following types of functions:

- Comparison, control flow, and cast functions that allow you to compare and convert data
- String, numeric, and date/time functions that allow you to manipulate, calculate, convert, extract, and concatenate data
- Aggregate functions that you can use in `SELECT` statements to summarize data that has been grouped together by a `GROUP BY` clause
- Encryption, system-related, and query and insert functions that allow you to perform system operations

Comparing and Converting Data

MySQL provides three types of functions that allow you to compare and convert data. These include comparison functions, control flow functions, and cast functions. In this section, you learn about these types of functions and review examples that demonstrate how they're used.

The functions included as part of the MySQL installation are referred to as built-in functions. These are the type of functions that you learn about in this chapter. MySQL also supports the creation of user-defined functions, which are functions that you can write in C or C++ and then use in MySQL SQL statements. To use user-defined functions, you must compile the mysqld server program dynamically, not statically, and your operating system must support dynamic loading. Because user-defined functions are beyond scope of this book, you should refer to the MySQL product documentation for more information about how to create and use them.

Comparison Functions

Comparison functions are similar to the comparison operators that you saw in Chapter 8. These functions allow you to compare different values and, from those comparisons, return one of the values or return a condition of true, false, or `NULL`. If either argument or both arguments in a comparison are `NULL`, `NULL` is returned. This section covers many of the more common comparison functions supported by MySQL.

GREATEST() and LEAST() Functions

Two functions that are useful for comparing values are the `GREATEST()` and `LEAST()` functions, which allow you to compare two or more values and return the value that is either the highest or lowest, depending on the function used. The values specified can be numeric, string, or date/time values and are compared based on the current character set. The `GREATEST()` function uses the following syntax:

```
GREATEST(<value1>, <value2> [{, <value>}...])
```

When you use this function, you must specify at least two values, although you can specify as many additional values as necessary. As with most arguments in a function, the arguments are separated by commas.

One other thing to note about this and any function is that the arguments are enclosed by parentheses and the opening parenthesis follows directly after the function name. A space after the function name is not permitted. For example, a basic `SELECT` statement might use the `GREATEST()` function as follows:

```
SELECT GREATEST(4, 83, 0, 9, -3);
```

If you were to execute this statement, the value 83 would be returned because it is the highest number in the set of values. Of course, you do not need to use the `GREATEST()` function to see that the highest number is 83. The example here merely demonstrates the format used when including the `GREATEST()` function in your SQL statement. In actuality, you would probably pass the function arguments to the statement through your application or through variables. For example, suppose that you are working with a database that includes a table that lists books and another table that lists publishers. Now suppose that you had used `SELECT` statements to assign PublisherID values to several variables. You can then use the `GREATEST()` function to compare the values represented by those variables. For example, the following `SELECT` statement uses the `GREATEST()` function in the `WHERE` clause:

```
SELECT BookID, BookTitle, PublisherID
FROM Books
WHERE PublisherID=GREATEST(@id1, @id2, @id3, @id4, @id5)
ORDER BY BookTitle;
```

This statement retrieves only those books that have the highest PublisherID value. To retrieve those books, you don't have to know the PublisherID values because those values were assigned to the variables in a separate process.

For the purposes of this chapter, the majority of the function examples shown are based on the most basic `SELECT` statement, where only the `SELECT` clause is included. This is done for the sake of brevity and in order to cover as many functions as reasonably possible. Keep in mind, though, that you may find that these functions provide you with far greater value when used in other ways in your SQL statements, as this chapter's Try It Out sections demonstrate.

Now take a look at the `LEAST()` function. As the following syntax shows, the function is identical to the `GREATEST()` function except that the `LEAST` keyword is used rather than `GREATEST`:

```
LEAST(<value1>, <value2> [{, <value>}...])
```

Again, you must include at least two values and separate those values with a comma, as shown in the following example:

```
SELECT LEAST(4, 83, 0, 9, -3);
```

As you would expect, this statement returns a value of -3. If you specify string values, then the lowest value alphabetically (for example, a before b) is returned, and if you specify date/time values, the earliest date is returned.

COALESCE() and ISNULL() Functions

The `COALESCE()` function returns the first value in the list of arguments that is not `NULL`. If all values are `NULL`, then `NULL` is returned. The following syntax shows how this function is used:

```
COALESCE(<value> [{, <value>}...])
```

For the `COALESCE()` function, you must specify at least one value, although the function is more useful if multiple values are provided, as shown in the following example:

```
SELECT COALESCE(NULL, 2, NULL, 3);
```

In this case, the value 2 is returned because it is the first value that is not `NULL`. The `ISNULL()` function is also concerned with null values, although the output is not one of the specified values. Instead, `ISNULL()` returns a value of 1 if the expression evaluates to `NULL`; otherwise, the function returns a value of 0. The syntax for the `ISNULL()` function is as follows:

```
ISNULL(<expression>)
```

When using this function, you must specify an expression in the parentheses, as shown in the following example:

```
SELECT ISNULL(1*NULL);
```

The expression in this statement is `1*NULL`. As you recall, an expression evaluates to `NULL` if either argument is `NULL`. Because the expression evaluates to `NULL`, the `ISNULL()` function returns a value of 1, rather than 0.

A function such as `ISNULL`() can be handy when developing an application if your application includes conditional logic that needs to determine whether a value is `NULL` before taking an action. For example, suppose that you have a form that updates data in a table. You want the form to ask for a user's date of birth only if that user already hasn't provided a date of birth. You can use the `ISNULL()` function to check the appropriate column in the database table to determine whether a value exists. If a value of 1 is returned, then your application should ask the user for a date of birth. If a value of 0 is returned, the user should not be asked.

INTERVAL() and STRCMP() Functions

The `INTERVAL()` function compares the first integer listed as an argument to the integers that follow the first integer. The following syntax shows how to use this function:

```
INTERVAL(<integer1>, <integer2> [{, <integer>}...])
```

Starting with `<integer2>`, the values must be listed in ascending order. If `<integer1>` is less than `<integer2>`, a value of 0 is returned. If `<integer1>` is less than `<integer3>`, a value of 1 is returned. If `<integer1>` is less than `<integer4>`, a value of 2 is returned, and so on. The following `SELECT` statement demonstrates how the `INTERVAL()` function works:

```
SELECT INTERVAL(6, -2, 0, 4, 7, 10, 12);
```

In this case, `<integer1>` is greater than `<integer2>`, `<integer3>`, and `<integer4>`, but less than `<integer5>`, so a value of 3 is returned, which represents the position of `<integer5>`. In other words, 7 is the first value in the list that 6 is less than.

The `STRCMP()` is different from the `INTERVAL()` function in that it compares string values that can be literal values or derived from expressions, as shown in the following syntax:

```
STRCMP(<expression1>, <expression2>)
```

As the syntax shows, the `STRCMP()` function compares exactly two values. The function returns a 0 if `<expression1>` equals `<expression2>` and returns -1 if `<expression1>` is smaller than `<expression2>`. If `<expression1>` is larger than `<expression2>`, or if a `NULL` is returned by the comparison, the function returns a 1. For example, the following SQL statement compares two literal values:

```
SELECT STRCMP('big', 'bigger');
```

The values are compared based on the current character set. Because big is smaller than bigger (it is first alphabetically), the statement returns a -1.

In the following exercise, you create two `SELECT` statements that each define a variable based on a DVDID value in the DVDs table of the DVDRentals database. Once you've assigned values to the variables, you include them in a `SELECT` statement that uses the `LEAST()` comparison function to compare values stored in the variables.

Try It Out Passing Values to a Comparison Function

The following steps describe how to create statements that use comparison functions:

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 assigns a value to the `@dvd1` variable. Execute the following SQL statement at the mysql command prompt:

```
SELECT @dvd1:=DVDID
FROM DVDs
WHERE DVDName='White Christmas';
```

   You should receive results similar to the following:

```
+--------------+
| @dvd1:=DVDID |
+--------------+
|            1 |
+--------------+
1 row in set (0.05 sec)
```

3. Next assign a value to the `@dvd2` variable. Execute the following SQL statement at the mysql command prompt:

```
SELECT @dvd2:=DVDID
FROM DVDs
WHERE DVDName='Out of Africa';
```

   You should receive results similar to the following:

```
+--------------+
| @dvd2:=DVDID |
+--------------+
|            3 |
+--------------+
1 row in set (0.00 sec)
```

4. Now use the variables that you created in steps 2 and 3 in the `WHERE` clause of a `SELECT` statement. Execute the following SQL statement at the mysql command prompt:

```
SELECT OrderID, TransID, DVDID
FROM Transactions
WHERE DVDID=LEAST(@dvd1, @dvd2)
ORDER BY OrderID, TransID;
```

You should receive results similar to the following:

```
+---------+---------+-------+
| OrderID | TransID | DVDID |
+---------+---------+-------+
|       1 |       1 |     1 |
|       3 |       6 |     1 |
|       6 |      11 |     1 |
|       8 |      14 |     1 |
|       9 |      17 |     1 |
+---------+---------+-------+
7 rows in set (0.10 sec)
```

How It Works

In Chapter 7, you learned how to create SELECT statements that assigned values to variables. You were then able to use those variables in other SQL statements executed in the same user session. To complete this exercise, you created the same type of SELECT statements so that the LEAST() comparison function could use variables to compare values in the WHERE clause of the following SELECT statement:

```
SELECT OrderID, TransID, DVDID
FROM Transactions
WHERE DVDID=LEAST(@dvd1, @dvd2)
ORDER BY OrderID, TransID;
```

In this statement, @dvd1 and @dvd2 are used as arguments in the LEAST() function. Because you assigned the @dvd1 variable a value of 1 and the @dvd2 variable a value of 3, the LEAST() function compares the value of 1 to the value of 3. As a result, the WHERE clause expression is interpreted as DVDID=1 because 1 is the lower of the two values. The rest of the SELECT statement is then executed as other SELECT statements you have seen. The SELECT list indicates that the result set should include only the OrderID, TransID, and DVDID columns. The WHERE clause indicates that the only rows returned should be those that contain a DVDID value of 1. The ORDER BY clause indicates that the result set should be ordered first by the values in the OrderID column and then by the values in the TransID column.

Control Flow Functions

The types of functions that you look at next are those that return a result by comparing conditions. The returned value is determined by which condition is true.

IF() Function

The IF() function compares three expressions, as shown in the following syntax:

```
IF(<expression1>, <expression2>, <expression3)
```

If <expression1> evaluates to true, then the function returns <expression2>; otherwise, the function returns <expression3>. Take a look at an example to demonstrate how this works. The following SELECT statement evaluates the first expression and then returns one of the two literal values:

```
SELECT IF(10>20, 'expression correct', 'expression incorrect');
```

In this statement, literal values are used for the second and third expressions. Because the first expression (10>20) evaluates to false, the third expression is returned. This function is useful if you want to return a specific response based on the value returned by the IF() function. For example, suppose that you're developing an application for a bookstore. The application should be able to check the database to determine whether requested books are in stock. If the book is in stock, the results should return the department in which the book can be found. If the book is not in stock, the results should return the International Standard Book Number (ISBN) of the book so that it can be ordered. You can use the IF() function to check a column that retrieves the number of books in stock. Based on the value returned, the function can then return either a value from a column that specifies the department where the book can be found or a value from a different column that specifies the book's ISBN.

IFNULL() and NULLIF() Functions

The IFNULL() function returns a value based on whether a specified expression evaluates to NULL. The function includes two expressions, as shown in the following syntax:

```
IFNULL(<expression1>, <expression2>)
```

The function returns <expression1> if it is not NULL; otherwise, it returns <expression2>. For example, the following SELECT statement includes an IFNULL() function whose first expression is NULL:

```
SELECT IFNULL(10*NULL, 'expression incorrect');
```

Because the first expression (10*NULL) evaluates to NULL, the NULL value is not returned. Instead, the second expression is returned. In this case, the second expression is a literal value, so that is the value returned.

To get a better idea of how this function works, suppose that you are developing an application that tracks profile information about a company's customers. The profile data is stored in a database that includes a table for contact information. The table includes a column for a home phone number and a column for a cell phone number. Along with other details about the employee, the application should display the customer's home phone number if that is known. If not, the application should display the cell phone number. You can use the IFNULL() function to specify that the home number should be returned unless that value is NULL, in which case the cell phone number should be returned.

The NULLIF() function is a little different from the IFNULL() function. The NULLIF() function returns NULL if <expression1> equals <expression2>; otherwise, it returns <expression1>. The syntax for the NULLIF() function is as follows:

```
NULLIF(<expression1>, <expression2>)
```

The following SELECT statement demonstrates how this works:

```
SELECT NULLIF(10*20, 20*10);
```

As you can see, the statement specifies two expressions. Because they are equal (they both return a value of 200), NULL is returned, rather than the value of 200 returned by the first expression.

CASE() Function

The CASE() function is a little more complicated than the previous control flow functions that you looked at. This function, though, provides far more flexibility in terms of the number of conditions that you can evaluate and the type of results that you can provide.

The CASE() function supports two slightly different formats. The first of these is shown in the following syntax:

```
CASE WHEN <expression> THEN <result>
   [{WHEN <expression> THEN <result>}...]
   [ELSE <result>]
END
```

As the syntax shows, you must specify the CASE keyword, followed by at least one WHEN...THEN clause. The WHEN...THEN clause specifies the expression to be evaluated and the results to be returned if that expression evaluates to true. You can specify as many WHEN...THEN clauses as necessary. The next clause is the ELSE clause, which is also optional. The ELSE clause provides a default result in case none of the expressions in the WHEN...THEN clauses evaluate to true. Finally, the CASE() function construction must be terminated with the END keyword.

The following SELECT statement demonstrates how this form of the CASE() function works:

```
SELECT CASE WHEN 10*2=30 THEN '30 correct'
   WHEN 10*2=40 THEN '40 correct'
   ELSE 'Should be 10*2=20'
END;
```

As you can see, the first WHEN...THEN clause specifies an expression (10*2=30) and a result ('30 correct'). The result is returned if the expression evaluates to true. The statement includes two WHEN...THEN clauses; if the expression in the first one evaluates to true, that result is returned. If the expression in the second WHEN...THEN clause evaluates to true, that result is returned. If neither clause evaluates to true, the result in the ELSE clause is returned. In this case, the result in the ELSE clause is returned because neither WHEN...THEN clause expression evaluates to true.

The next version of the CASE() function is slightly different from the first, as shown in the following syntax:

```
CASE <expression>
   WHEN <value> THEN <result>
   [{WHEN <value> THEN <result>}...]
   [ELSE <result>]
END
```

The main difference in this version of the CASE() function is that the expression is specified after the keyword CASE, and the WHEN...THEN clauses include the possible values that result from that expression. The following example demonstrates how this form of the CASE() function works:

```
SELECT CASE 10*2
   WHEN 20 THEN '20 correct'
   WHEN 30 THEN '30 correct'
   WHEN 40 THEN '40 correct'
END;
```

As you can see, the `CASE()` function includes the expression (`10*2`) after the `CASE` keyword. Three `WHEN...THEN` clauses follow the `CASE` keyword, each of which contains a possible value that represents the value returned by the expression. In this case, the first `WHEN...THEN` clause evaluates to true because it contains the correct value (`20`) returned by the expression. Also notice that the `CASE()` function doesn't include an `ELSE` clause. The assumption here is that one of the `WHEN...THEN` clauses contains the correct values. If you do include an `ELSE` clause, it works the same way as the `ELSE` clause in the previous form of the `CASE()` function. The `ELSE` clause provides a default result in case none of the expressions in the `WHEN...THEN` clauses evaluates to true.

The `CASE()` function is similar to the logic found in application languages. By using the `CASE()`, you're switching the conditional logic from the application side to the database side. This can be useful if you want to shift some of the business logic to the back end, helping to minimize some of the front-end processing. In this way, the database determines what value to return, rather than the application itself.

For example, suppose that you are developing an application that retrieves data about different car models. If a model is available in different editions, a list of the editions should be returned, as well as the basic details about the model; otherwise, only basic details should be returned. You can use the `CASE()` function to first determine how many editions are available for a model. If the amount is greater than one, then the list of editions should be included in the results; otherwise, no edition information should be included.

In the next exercise you try out some of the control flow functions that you learned about in this section. To use these functions, you create `SELECT` statements that retrieve data from the DVDs table in the DVDRentals database.

Try It Out Using Control Flow Functions in a SELECT Statement

The following steps describe how to create statements that use control flow functions:

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 uses the `IF()` function in the `SELECT` clause to retrieve data. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName AS Title, StatID AS Status, RatingID AS Rating,
   IF(NumDisks>1, 'Check for extra disks!', 'Only 1 disk.') AS Verify
FROM DVDs
ORDER BY Title;
```

   You should receive results similar to the following:

```
+------------------------------+--------+--------+------------------------+
| Title                        | Status | Rating | Verify                 |
+------------------------------+--------+--------+------------------------+
| A Room with a View           | s1     | NR     | Only 1 disk.           |
| Amadeus                      | s2     | PG     | Only 1 disk.           |
| Mash                         | s2     | R      | Check for extra disks! |
| Out of Africa                | s1     | PG     | Only 1 disk.           |
| The Maltese Falcon           | s2     | NR     | Only 1 disk.           |
| The Rocky Horror Picture Show | s2    | NR     | Check for extra disks! |
| What's Up, Doc?                | s2     | G      | Only 1 disk.           |
| White Christmas                | s1     | NR     | Only 1 disk.           |
+--------------------------------+--------+--------+------------------------+
8 rows in set (0.00 sec)
```

3. Now try out the `CASE()` function. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, RatingID AS Rating,
   CASE
      WHEN RatingID='R' THEN 'Under 17 requires an adult.'
      WHEN RatingID='X' THEN 'No one 17 and under.'
      WHEN RatingID='NR' THEN 'Use discretion when renting.'
      ELSE 'OK to rent to minors.'
   END AS Policy
FROM DVDs
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+--------------------------------+--------+------------------------------+
| DVDName                        | Rating | Policy                       |
+--------------------------------+--------+------------------------------+
| A Room with a View             | NR     | Use discretion when renting. |
| Amadeus                        | PG     | OK to rent to minors.        |
| Mash                           | R      | Under 17 requires an adult.  |
| Out of Africa                  | PG     | OK to rent to minors.        |
| The Maltese Falcon             | NR     | Use discretion when renting. |
| The Rocky Horror Picture Show  | NR     | Use discretion when renting. |
| What's Up, Doc?                | G      | OK to rent to minors.        |
| White Christmas                | NR     | Use discretion when renting. |
+--------------------------------+--------+------------------------------+
8 rows in set (0.01 sec)
```

4. In the next statement, you also use the `CASE()` function, but you use a different form of that function. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, RatingID AS Rating,
   CASE RatingID
      WHEN 'R' THEN 'Under 17 requires an adult.'
      WHEN 'X' THEN 'No one 17 and under.'
      WHEN 'NR' THEN 'Use discretion when renting.'
      ELSE 'OK to rent to minors.'
   END AS Policy
FROM DVDs
ORDER BY DVDName;
```

You should receive the same results that you received when you executed the statement in Step 3.

How It Works

You should be well familiar with most of the elements in the `SELECT` statement that you created in this exercise. If you have any questions about how the statement is used, refer back to Chapter 7 for an explanation of `SELECT` statements. These statements include functions that you have not used before, so individual examinations of each one follow. In the first `SELECT` statement that you created, you included the `IF()` function:

```
SELECT DVDName AS Title, StatID AS Status, RatingID AS Rating,
   IF(NumDisks>1, 'Check for extra disks!', 'Only 1 disk.') AS Verify
FROM DVDs
ORDER BY Title;
```

The `IF()` function is used as one of the elements in the `SELECT` list. The function includes three arguments. The first argument is an expression (`NumDisks>1`) that determines whether the second or third argument in the function is returned. For each row in which the NumDisks value is greater than 1, the first message is returned. For all other rows, the second message is returned. Notice that the column in the result set that includes the data returned by the `IF()` function is named Verify. You can assign aliases to select list elements constructed with functions as you would any other select list element. (For details about assigning an alias to an element in the select list, see Chapter 7.)

The next `SELECT` statement that you created uses a `CASE()` function to specify a set of results to return depending on the value in the RatingID column:

```
SELECT DVDName, RatingID AS Rating,
   CASE
      WHEN RatingID='R' THEN 'Under 17 requires an adult.'
      WHEN RatingID='X' THEN 'No one 17 and under.'
      WHEN RatingID='NR' THEN 'Use discretion when renting.'
      ELSE 'OK to rent to minors.'
   END AS Policy
FROM DVDs
ORDER BY DVDName;
```

The `CASE()` function includes three `WHEN...THEN` clauses and one `ELSE` clause. This means that the function can return four possible results, depending on the value in the RatingID column. Specific results are assigned to the values `R`, `X`, and `NR`, but all other RatingID values are assigned the result in the `ELSE` clause. Notice that the `CASE()` function is one of the elements in the select list of the `SELECT` clause. Also notice that it is assigned the name Policy. For each row returned by the `SELECT` statement, the Policy column includes one of the four results returned by the `CASE()` function.

The last `SELECT` statement that you created in this exercise uses a different form of the `CASE()` function than you used in the previous statement; however, the outcome is the same. In the updated `SELECT` statement, the expression is specified after the `CASE` keyword:

```
SELECT DVDName, RatingID AS Rating,
   CASE RatingID
      WHEN 'R' THEN 'Under 17 requires an adult.'
      WHEN 'X' THEN 'No one 17 and under.'
      WHEN 'NR' THEN 'Use discretion when renting.'
      ELSE 'OK to rent to minors.'
   END AS Policy
FROM DVDs
ORDER BY DVDName;
```

As you can see, the expression specified after `CASE` is made up only of the RatingID column name. No other elements are necessary to qualify as an expression. In addition, the `WHEN...THEN` clauses no longer contain expressions, but contain only the values returned by the RatingID column.

Cast Functions

Cast functions allow you to convert values to a specific type of data or to assign a character set to a value. The first of these functions is the CAST() function, which is shown in the following syntax:

```
CAST(<expression> AS <type>)
```

The function converts the value returned by the expression to the specified conversion type, which follows the AS keyword. The CAST() function supports a limited number of conversion types. These types are similar to data types, but they are specific to the CAST() function (and the CONVERT() function) and serve a slightly different purpose, which is to specify how the data is converted. Data types, on the other hand, specify the type of data that can be inserted in a column. (For more information about data types, see Chapter 5.)

The conversion types available to the CAST() function are as follows:

- BINARY
- CHAR
- DATE
- DATETIME
- SIGNED [INTEGER]
- TIME
- UNSIGNED [INTEGER]

For example, you might have a numeric value (either a literal value or one returned by an expression) that you want converted to the DATE conversion type. The following SELECT statement demonstrates how you might do this:

```
SELECT CAST(20041031 AS DATE);
```

As you can see, the value is specified, followed by the AS keyword, and then followed by the DATE conversion type. The value returned by this function will be in a DATE format (2004-10-31).

The CONVERT() function allows you to convert dates in the same way as the CAST() function, only the format is a little different, as shown in the following syntax:

```
CONVERT(<expression>, <type>)
```

Notice that you need to specify only the expression and the conversion type, without the AS keyword, but you must separate the two by a comma. The conversion types you can use in the CONVERT() function are the same as those you can use for the CAST() function. For example, the following SELECT statement produces the same results as the last example:

```
SELECT CONVERT(20041031, DATE);
```

Notice that you need to specify only the numeric value and the DATE conversion type. The CONVERT() function, however, also includes another form, as shown in the following syntax:

```
CONVERT(<expression> USING <character set>)
```

This form is used to assign a character set to the specified expression. For example, the following SELECT statement converts a string to the latin2 character set:

```
SELECT CONVERT('cats and dogs' USING latin2);
```

In this statement, the CONVERT() function includes the expression (which is a literal string), followed by the USING keyword and the name of the character set (latin2). The value returned is the string value in the new character set. (In this case, the value is the same as it appears in the function.)

The CAST() and CONVERT() functions are particularly useful when you want to convert data stored in a MySQL database to a format that can be used by an application. For example, suppose that you have a table in a MySQL database that includes a DATETIME column. You also have an application that needs to use those values, but it cannot work with them as DATETIME values. As a result, you need to convert those values to numerical (UNSIGNED INTEGER) values that can be used by the application. To achieve this conversion, you can use the CAST() or CONVERT() function as you're retrieving data from the database. This next exercise allows you to try out the CAST() and CONVERT() functions in SELECT statements. You convert a date value to an integer.

Try It Out Converting Data to Different Types

To create statements that convert values, 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 includes the CAST() function. Execute the following SQL statement at the mysql command prompt:

```
SELECT OrderID, TransID, DVDID,
   CAST(DateOut AS UNSIGNED INTEGER) AS DateOut_INT
FROM Transactions
WHERE DVDID=4 OR DVDID=5 OR DVDID=7
ORDER BY OrderID, TransID, DVDID;
```

You should receive results similar to the following:

```
+---------+---------+-------+-------------+
| OrderID | TransID | DVDID | DateOut_INT |
+---------+---------+-------+-------------+
|       1 |       2 |     4 |    20041007 |
|       3 |       5 |     4 |    20041007 |
|       3 |       7 |     7 |    20041007 |
|       4 |       8 |     4 |    20041007 |
|       7 |      12 |     4 |    20041007 |
|       9 |      16 |     7 |    20041007 |
|      10 |      18 |     5 |    20041007 |
|      12 |      22 |     5 |    20041007 |
|      13 |      23 |     7 |    20041007 |
+---------+---------+-------+-------------+
9 rows in set (0.10 sec)
```

3. Next modify the SELECT statement to use the CONVERT() function. Execute the following SQL statement at the mysql command prompt:

```
SELECT OrderID, TransID, DVDID,
   CONVERT(DateOut, UNSIGNED) AS DateOut_INT
FROM Transactions
WHERE DVDID=4 OR DVDID=5 OR DVDID=7
ORDER BY OrderID, TransID, DVDID;
```

You should receive the same results that you received when you executed the SELECT statement in Step 2.

How It Works

In this exercise, you created two SELECT statements. The first of these included the CAST() function as an element in the select list of the SELECT clause:

```
SELECT OrderID, TransID, DVDID,
   CAST(DateOut AS UNSIGNED INTEGER) AS DateOut_INT
FROM Transactions
WHERE DVDID=4 OR DVDID=5 OR DVDID=7
ORDER BY OrderID, TransID, DVDID;
```

The CAST() function extracts the values from the DateOut column in the Transactions table and converts them to UNSIGNED INTEGER values. The converted values are included in the result set in the DateOut_INT column.

The next SELECT statement that you created produces the same results as the first, only this time you used the CONVERT() function:

```
SELECT OrderID, TransID, DVDID,
   CONVERT(DateOut, UNSIGNED) AS DateOut_INT
FROM Transactions
WHERE DVDID=4 OR DVDID=5 OR DVDID=7
ORDER BY OrderID, TransID, DVDID;
```

As you can see, the DateOut column is again specified as the first argument; however, the second argument consists only of the keyword UNSIGNED. When you specify UNSIGNED, it is the same as specifying UNSIGNED INTEGER. You do not need to include the keyword INTEGER in this case.

Managing Different Types of Data

MySQL includes functions that allow you to manage string, numeric, and date/time data. Unlike the functions that you've already looked at, the next set of functions is specific to a type of data. You can use these functions in conjunction with the functions you've already seen. In many cases, a function can be embedded as an argument in other functions, which makes the use of functions all the more powerful. In several of the Try It Out sections that follow, you see how to embed functions as arguments in other functions.

String Functions

As you would guess, string functions allow you to manipulate and extract string values. MySQL supports numerous string functions. This section covers those that you're most likely to use in your applications and provides examples of each of them.

ASCII() and ORD() Functions

The `ASCII()` function allows you to identify the numeric value of the first character in a string. The syntax for the function is as follows:

```
ASCII(<string>)
```

To use the `ASCII()` function, you need only to identify the string, as shown in the following example:

```
SELECT ASCII('book');
```

The `SELECT` statement returns the numeric value for the first character, which is the letter b. The numeric value for the letter b is 98.

The `ASCII()` function works only for single-byte characters (with values from 0 to 255). For multi-byte characters, you should use the `ORD()` function, which is shown in the following syntax:

```
ORD(<string>)
```

The `ORD()` function works just like the `ASCII()` function except that it also supports multibyte characters. To use the `ORD()` function, specify a string (which can include numerals), as shown in the following example:

```
SELECT ORD(37);
```

As with the `ASCII()` function, the `ORD()` function returns the numeric value of the first character. For the number 3, the numeric value is 51. If you specify a number, rather than a regular string, you do not need to include the single quotes. In addition, if the function argument is a single-byte character, the results are the same as what you would see when using the `ASCII()` function.

CHAR_LENGTH(), CHARACTER_LENGTH(), and LENGTH() Functions

The `CHAR_LENGTH()` and `CHARACTER_LENGTH()` functions, which are synonymous, return the number of characters in the specified string. The following syntax shows how to use either function:

```
CHAR_LENGTH(<string>)
```

As you can see, you need only to specify the string to determine the length of that string, as shown in the following example:

```
SELECT CHAR_LENGTH('cats and dogs');
```

The statement returns a value of 13, which is the number of characters in the string, including spaces.

The LENGTH() function also returns the length of a string, only the length is measured in bytes, rather than characters. The syntax for the LENGTH() function is similar to the CHAR_LENGTH() and CHARACTER_LENGTH functions:

```
LENGTH(<string>)
```

If you use the LENGTH() function with single-byte characters, the results are the same as with the CHAR_LENGTH() function, as shown in the following example:

```
SELECT LENGTH('cats and dogs');
```

In this case, the result is once again 13. If this were a double-byte character string, though, the result would be 26 because the LENGTH() function measures in bytes, not characters.

The CHAR_LENGTH() and LENGTH() functions can be useful if you need to return values that are greater than or less than a specific length. For example, suppose that you have a table that includes a list of pet-related products. The table includes a Description column that provides a brief description of each product. You want all descriptions to be no longer than 50 characters; however, some descriptions exceed this amount, so you plan to cut content in those columns. To determine which columns exceed 50 characters, you can use the CHAR_LENGTH() function to retrieve those rows whose Description value is greater than 50 characters.

CHARSET() and COLLATION() Functions

The CHARSET() function identifies the character set used for a specified string, as shown in the following syntax:

```
CHARSET(<string>)
```

For example, the following SELECT statement uses the CHARSET() function to return the character set used for the 'cats and dogs' string:

```
SELECT CHARSET('cats and dogs');
```

If you are running a default installation of MySQL, the SELECT statement returns a value of latin1.

You can also identify the collation used for a string by using the COLLATION() function, shown in the following syntax:

```
COLLATION(<string>)
```

As with the CHARSET() function, you need only to specify the string, as the following SELECT statement demonstrates:

```
SELECT COLLATION('cats and dogs');
```

In this case, if working with a default installation of MySQL, the SELECT statement returns a value of latin1_swedish_ci.

Using the CHARSET() and COLLATION() functions to identify the character set or collation of a string can be useful when you want to find this information quickly, without having to search column, table, database, and system settings. By simply using the appropriate function when you retrieve the data, you can avoid the possibility of having to take numerous steps to find the information you need, allowing you to determine exactly what you need with one easy step.

CONCAT() and CONCAT_WS() Functions

MySQL provides two very useful functions that allow you to concatenate data. The first of these is the `CONCAT()` function, which is shown in the following syntax:

```
CONCAT(<string1>, <string2> [{, <string>}...])
```

As the syntax demonstrates, you must specify two or more string values, which are separated by commas. For example, the following statement concatenates five values:

```
SELECT CONCAT('cats', ' ', 'and', ' ', 'dogs');
```

Notice that the second and fourth values are spaces. This ensures that a space is provided between each of the three words. As a result, the output from this function (`cats and dogs`) is shown correctly. Another way you can include the spaces is by using the `CONCAT_WS()` function, which allows you to define a separator as one of the arguments in the function, as shown in the following syntax:

```
CONCAT_WS(<separator>, <string1>, <string2> [{, <string>}...])
```

By using this function, the separator is automatically inserted between the values. If one of the values is `NULL`, the separator is not used. Except for the separator, the `CONCAT_WS()` function is the same as the `CONCAT()` function. For example, the following `SELECT` statement concatenates the same words as in the last example:

```
SELECT CONCAT_WS(' ', 'cats', 'and', 'dogs');
```

Notice that the `CONCAT_WS()` function identifies the separator (a space) in the first argument and that the separator is followed by the string values to be concatenated. The output from this function (cats and dogs) is the same as the output you saw in the `CONCAT()` example.

The `CONCAT()` and `CONCAT_WS` functions can be useful in a number of situations. For example, suppose that you have a table that displays employee first names and last names in separate columns. You can use one of these functions to display the names in a single column, while still sorting them according to the last names. You can also use the functions to join other types of data, such as a color to a car model (for instance, red Honda) or a flavor to a food (for instance, chocolate ice cream). There are no limits to the types of string data that you can put together.

INSTR() and LOCATE() Functions

The `INSTR()` function takes two arguments, a string and a substring, as shown in the following syntax:

```
INSTR(<string>, <substring>)
```

The function identifies where the substring is located in the string and returns the position number. For example, the following `INSTR()` function returns the position of `dogs` in the string:

```
SELECT INSTR('cats and dogs', 'dogs');
```

In this case, the substring dogs begins in the tenth position, so the function returns a value of 10. You can achieve the same results by using the LOCATE() function, shown in the following syntax:

```
LOCATE(<substring>, <string>)
```

As you can see, the syntax for the LOCATE() and INSTR() functions is similar except that, with LOCATE(), the substring is listed first, as shown in the following example:

```
SELECT LOCATE('dogs', 'cats and dogs');
```

Again, the function returns a value of 10. The LOCATE() function, however, provides another alternative, as the following syntax demonstrates:

```
LOCATE(<substring>, <string>, <position>)
```

The function includes a third argument, <position>, which identifies a starting position in the function. This is the position at which the function should start looking for the substring. For example, suppose that you create the following SELECT statement:

```
SELECT LOCATE('dogs', 'cats and dogs and more dogs', 15);
```

Notice that the LOCATE() function includes a third argument: 15. This is the position at which the function should begin looking for the substring dogs. As a result, the function disregards the first occurrence of dogs because it is before position 15 and returns a value of 24, which is where the second dogs begins.

LCASE(), LOWER(), UCASE(), and UPPER() Functions

MySQL also includes functions that allow you to change string values to upper or lowercase. For example, the LCASE() and LOWER() functions, which are synonymous, change the case of the specified string, as shown in the following syntax:

```
LOWER(<string>)
```

As you can see, you need to include the string as a function argument. For example, the following SELECT statement uses the LOWER() function to remove the initial capitalizations from the string:

```
SELECT LOWER('Cats and Dogs');
```

The output from this statement is cats and dogs. Notice that the string value now includes no uppercase letters. You can also change lowercase to uppercase by using the UPPER() or UCASE() functions, which are also synonymous. The following syntax shows the UPPER() function:

```
UPPER(<string>)
```

Notice that, as with the LOWER() function, you need only to supply the string, as shown in the following example:

```
SELECT UPPER('cats and dogs');
```

By using the UPPER() function, all characters in the string are returned as uppercase (CATS AND DOGS).

The LOWER() and UPPER() functions are useful whenever you need to change the case of a value. For example, suppose that you have an application that allows customers to register online for product information. The registration information is stored in a table in your database. When the customers enter their e-mail addresses, the addresses are stored as they are entered by the customers, so some are stored as all lowercase, some all uppercase, and some mixed case. When you retrieve those e-mail addresses, you want them displayed as all lowercase. You can use the LOWER() function when you retrieve the e-mail addresses to provide a uniform display.

LEFT() and RIGHT() Functions

MySQL also provides functions that return only a part of a string value. For example, you can use the LEFT() function to return only a specific number of characters from a value, as shown in the following syntax:

```
LEFT(<string>, <length>)
```

The <length> value determines how many characters are returned, starting at the left end of the string. For example, the following SELECT statement returns only the first four characters of the string:

```
SELECT LEFT('cats and dogs', 4);
```

Because the value 4 is specified in the function arguments, the function returns the value cats.

You can also specify which characters are returned starting at the right end of the string by using the following RIGHT() function:

```
RIGHT(<string>, <length>)
```

Notice that the syntax is similar to the LEFT() function. You must again specify the length of the substring that is returned. For example, the following SELECT statement returns only the last four characters of the specified string:

```
SELECT RIGHT('cats and dogs', 4);
```

In this case, the statement returns the value dogs.

The LEFT() and RIGHT() functions are useful when you want to use only part of values from multiple columns. For example, suppose that you want to create a user ID for your employees based on their first and last names. You can use the LEFT() function to take the first three letters of their first names and the first four letters of their last names and then use the CONCAT() function to join these extracted values together.

REPEAT() and REVERSE() Functions

The REPEAT() function, shown in the following syntax, is used to repeat a string a specific number of times:

```
REPEAT(<string>, <count>)
```

To use this function, you must first specify the string and then the number of times that the string should be repeated. The values are then concatenated and returned. For example, the following SELECT statement uses the REPEAT() function to repeat CatsDogs three times:

```
SELECT REPEAT('CatsDogs', 3);
```

The result from this function is CatsDogsCatsDogsCatsDogs.

In addition to repeating string values, you can reverse their order by using the following REVERSE() function:

```
REVERSE(<string>)
```

In this case, you need to specify only the string, as the following SELECT statement shows:

```
SELECT REVERSE('dog');
```

The value returned by this function is god, which, as anyone with a dog will tell you, is exactly what you should expect.

SUBSTRING() Function

The final string function that you examine in this section is the SUBSTRING() function. The function, which includes several forms, returns a substring from the identified string. The first form of the SUBSTRING() function is shown in the following syntax:

```
SUBSTRING(<string>, <position>)
```

In this form of the SUBSTRING() function, you must specify the string and the starting position. The function then returns a substring that includes the rest of the string value, starting at the identified position. You can achieve the same results by using the following syntax:

```
SUBSTRING(<string> FROM <position>)
```

In this case, you must separate the two arguments with the FROM keyword, rather than a comma; however, either method works. For example, you can use the following SELECT statement to return the substring dog, which starts at the tenth position.

```
SELECT SUBSTRING('cats and dogs', 10);
```

As you might have noticed, the SUBSTRING() function, when used this way, is a little limiting because it provides only a starting position but no ending position. MySQL does support another form of the SUBSTRING() function:

```
SUBSTRING(<string>, <position>, <length>)
```

This form includes the <length> argument, which allows you to specify how long (in characters) the substring should be. You can also use the following format to specify the length:

```
SUBSTRING(<string> FROM <position> FOR <length>)
```

In this case, instead of using commas to separate the arguments, you use the FROM and FOR keywords, but the results are the same. The following example demonstrates how to specify a length:

```
SELECT SUBSTRING('cats and dogs and more dogs', 10, 4);
```

Notice that, after defining the string, the function arguments identify the starting position (10) and the length of the substring (4). As a result, the function returns a value of dogs.

As you have seen, string functions allow you to manipulate and extract string values. In the following Try It Out, you try out several of these functions by creating SELECT statements that retrieve data from the DVDRentals database.

Try It Out Using String Functions in Your SQL Statements

To create statements employing string functions, 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 statement that you create uses the CHAR_LENGTH() function to specify the length of a retrieved value. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, CHAR_LENGTH(DVDName) AS CharLength
FROM DVDs
WHERE CHAR_LENGTH(DVDName)>10
ORDER BY DVDName;
```

You should receive results similar to the following:

```
+-------------------------------+------------+
| DVDName                       | CharLength |
+-------------------------------+------------+
| A Room with a View            |         18 |
| Out of Africa                 |         13 |
| The Maltese Falcon            |         18 |
| The Rocky Horror Picture Show |         29 |
| What's Up, Doc?               |         15 |
| White Christmas               |         15 |
+-------------------------------+------------+
6 rows in set (0.00 sec)
```

3. The next SELECT statement that you create uses the CONCAT_WS() function to concatenate the employee names. Execute the following SQL statement at the mysql command prompt:

```
SELECT EmpID, CONCAT_WS(' ', EmpFN, EmpMN, EmpLN) AS Name
FROM Employees
ORDER BY EmpLN;
```

You should receive results similar to the following:

```
+-------+---------------------+
| EmpID | Name                |
+-------+---------------------+
|     6 | George Brooks       |
|     5 | Rita C. Carter      |
|     4 | John Laguci         |
|     3 | Mary Marie Michaels |
|     2 | Robert Schroader    |
|     1 | John P. Smith       |
+-------+---------------------+
6 rows in set (0.00 sec)
```

4. Next, use the `CONCAT()` and `LEFT()` functions to create registration codes for the employees. Execute the following SQL statement at the mysql command prompt:

```
SELECT EmpID, CONCAT(LEFT(EmpFN, 2), LEFT(EmpLN, 3), EmpID) AS RegID
FROM Employees
ORDER BY EmpID;
```

You should receive results similar to the following:

```
+-------+--------+
| EmpID | RegID  |
+-------+--------+
|     1 | JoSmi1 |
|     2 | RoSch2 |
|     3 | MaMic3 |
|     4 | JoLag4 |
|     5 | RiCar5 |
|     6 | GeBro6 |
+-------+--------+
6 rows in set (0.00 sec)
```

5. Now create a `SELECT` statement that uses the `UPPER()`, `CONCAT_WS()`, `CONCAT()`, `LOWER()`, and `LEFT()` functions to manipulate retrieved data from the Employees table. Execute the following SQL statement at the mysql command prompt:

```
SELECT EmpID, UPPER(CONCAT_WS(' ', EmpFN, EmpMN, EmpLN)) AS Name,
   CONCAT(LOWER(LEFT(EmpFN, 2)), LOWER(LEFT(EmpLN, 3)), EmpID) AS RegID
FROM Employees
ORDER BY EmpID;
```

You should receive results similar to the following:

```
+-------+---------------------+--------+
| EmpID | Name                | RegID  |
+-------+---------------------+--------+
|     1 | JOHN P. SMITH       | josmi1 |
|     2 | ROBERT SCHROADER    | rosch2 |
|     3 | MARY MARIE MICHAELS | mamic3 |
|     4 | JOHN LAGUCI         | jolag4 |
|     5 | RITA C. CARTER      | ricar5 |
|     6 | GEORGE BROOKS       | gebro6 |
+-------+---------------------+--------+
6 rows in set (0.00 sec)
```

6. Your final SELECT statement includes the UPPER(), CONCAT_WS(), CONCAT(), LOWER(), and SUBSTRING() functions to use on data from the Employees table. Execute the following SQL statement at the mysql command prompt:

```
SELECT EmpID, UPPER(CONCAT_WS(' ', EmpFN, EmpMN, EmpLN)) AS Name,
   CONCAT(LOWER(SUBSTRING(EmpFN, 2, 2)),
   LOWER(SUBSTRING(EmpLN, 2, 3)), EmpID) AS RegID
FROM Employees
ORDER BY EmpID;
```

You should receive results similar to the following:

```
+-------+--------------------+--------+
| EmpID | Name               | RegID  |
+-------+--------------------+--------+
|     1 | JOHN P. SMITH      | ohmit1 |
|     2 | ROBERT SCHROADER   | obchr2 |
|     3 | MARY MARIE MICHAELS | arich3 |
|     4 | JOHN LAGUCI        | ohagu4 |
|     5 | RITA C. CARTER     | itart5 |
|     6 | GEORGE BROOKS      | eoroo6 |
+-------+--------------------+--------+
6 rows in set (0.00 sec)
```

How It Works

The first statement that you created uses the CHAR_LENGTH() function, as shown in the following statement:

```
SELECT DVDName, CHAR_LENGTH(DVDName) AS CharLength
FROM DVDs
WHERE CHAR_LENGTH(DVDName)>10
ORDER BY DVDName;
```

The function appears as one of the elements of the select list in the SELECT clause and in an expression in the WHERE clause. The function retrieves the number of characters in a value. Notice that, in this case, the <string> argument is the name of the DVDName column. As a result, the string values are taken from that column. In addition, the WHERE clause specifies that the result set should include only those DVDName values with a length greater than 10.

The next statement that you created uses the CONCAT_WS() function to concatenate the employee names:

```
SELECT EmpID, CONCAT_WS(' ', EmpFN, EmpMN, EmpLN) AS Name
FROM Employees
ORDER BY EmpLN;
```

The function contains four arguments. The first argument is the separator, which in this case is a space. As a result, a space is added between all concatenated values, except those that are NULL. The next three arguments are the names of the columns from which the string values are extracted. For each row the SELECT statement returns, the employee's first name, middle name (in some cases), and last name appear together, with a space between values.

In the next statement, you used a `CONCAT()` function to concatenate the employee names and IDs. As the following statement shows, you used only part of the first and last names:

```
SELECT EmpID, CONCAT(LEFT(EmpFN, 2), LEFT(EmpLN, 3), EmpID) AS RegID
FROM Employees
ORDER BY EmpID;
```

The `LEFT()` function extracts only part of the first and last names. In the first usage, a 2 is specified as the second argument, so only the first two characters are retrieved from the first name. In the second usage of the `LEFT()` function, a 3 is specified as the second argument, so only the first three characters are retrieved. The retrieved characters are then concatenated to create one value made up of the first two letters of the first name, the first three letters of the last name, and the employee ID.

One thing to notice about the last statement is that you embedded one function as an argument within another function. You can embed functions to as many levels as necessary. For example, in the next statement that you created, you embedded functions to an additional level:

```
SELECT EmpID, UPPER(CONCAT_WS(' ', EmpFN, EmpMN, EmpLN)) AS Name,
   CONCAT(LOWER(LEFT(EmpFN, 2)), LOWER(LEFT(EmpLN, 3)), EmpID) AS RegID
FROM Employees
ORDER BY EmpID;
```

In the second element of the select clause, you used the familiar `CONCAT_WS()` construction to concatenate the employee name. Then you embedded the function as an argument in the `UPPER()` function so that the value returns in upper case. In the third element of the select list, you first used `LEFT()` functions to retrieve the first two letters from the first name and the first three letters from the last name. You then embedded each of these functions as an argument in a `LOWER()` function. As a result, the values retrieved by the `LEFT()` function changed to all lowercase. From these values, you used the `CONCAT()` function to concatenate the three arguments. As a result, one value is created that consists of the first two letters of the first name (in lowercase), the first three letters of the last name (in lowercase), and the employee ID.

The last `SELECT` statement that you created deviated from the previous statement by using a `SUBSTRING()` function rather than a `LEFT()` function:

```
SELECT EmpID, UPPER(CONCAT_WS(' ', EmpFN, EmpMN, EmpLN)) AS Name,
   CONCAT(LOWER(SUBSTRING(EmpFN, 2, 2)),
   LOWER(SUBSTRING(EmpLN, 2, 3)), EmpID) AS RegID
FROM Employees
ORDER BY EmpID;
```

Because you used the `SUBSTRING()` function, you can be more specific about which characters are retrieved from the first name and last name. For the first name, you retrieved two characters starting with the character in the second position. For the last name, you retrieved three characters starting with the character in the second position. The `SUBSTRING()` functions were then embedded in the `LOWER()` functions, which were then embedded as arguments for the `CONCAT()` function. The result is a value for each employee that is made up of the second and third letters of the first name (in lowercase), the second, third, and fourth letters of the last name (in lowercase), and the employee ID.

Numeric Functions

Now that you've had a good sampling of how string functions work, it's time to look at numeric functions. Numeric functions allow you to perform calculations on numeric values. MySQL supports various numeric functions that allow you to perform advanced mathematical operations. This section covers many of the more common numeric functions.

CEIL(), CEILING(), and FLOOR() Functions

The `CEIL()` and `CEILING()` functions, which are synonymous, return the smallest integer that is not less than the specified number. As the following syntax shows, to use this function, you need to specify only the number:

```
CEILING(<number>)
```

For example, the following `SELECT` statement returns a value of 10:

```
SELECT CEILING(9.327);
```

The value 10 is returned because it is the smallest integer that is not less than 9.327. However, if you want to retrieve the largest integer that is not greater than a specified value, you can use the `FLOOR()` function:

```
FLOOR(<number>)
```

The `FLOOR()` function is similar to the `CEILING()` function in the way that it is used. For example, the following `SELECT` statement uses the `FLOOR()` function:

```
SELECT FLOOR(9.327);
```

Notice that the same value is specified as in the previous example. Because the `FLOOR()` function retrieves the largest integer not greater than the specified value, a value of 9 is returned, rather than 10.

COT() Functions

MySQL includes numerous functions that allow you to calculate specific types of equations. For example, you can use the `COT()` function to determine the cotangent of a number:

```
COT(<number>)
```

As you can see, you need to provide only the number whose cotangent you want to find. For example, suppose you want to find the cotangent of 22. You can use the following `SELECT` statement:

```
SELECT COT(22);
```

MySQL returns a value of 112.97321035643, which is the cotangent of 22.

MOD() Function

The MOD() function is similar to the percentage (%) arithmetic operator you saw in Chapter 8. The function returns the remainder derived by dividing two numbers. The following syntax shows how to use a MOD() function:

```
MOD(<number1>, <number2>)
```

As you can see, you must specify the numbers that you want to divide as arguments, separated by a comma. The first argument that you specify is divided by the second argument, as shown in the following example.

```
SELECT MOD(22, 7);
```

In this statement, the MOD() function divides 22 by 7 and then returns the remainder. As a result, the function returns a value of 1.

PI() Function

The PI() function returns the value of PI. As the following syntax shows, you do not specify any arguments when using this function:

```
PI()
```

You can use the function to retrieve the value of PI to use in your SQL statement. At its simplest, you can use a basic SELECT statement to retrieve PI:

```
SELECT PI();
```

The PI() function returns a value of 3.141593.

POW() and POWER() Functions

The POW() and POWER() functions, which are synonymous, raise the value of one number to the power of the second number, as shown in the following syntax:

```
POW(<number>, <power>)
```

In the first argument, you must specify the root number. This is followed by the second argument (separated from the first by a comma) that specifies the power by which you should raise the root number. For example, the following SELECT statement raises the number 4 by the power of 2:

```
SELECT POW(4, 2);
```

The POW() function returns a value of 16.

ROUND() and TRUNCATE() Functions

There will no doubt be times when retrieving data from a MySQL database when you want to round off numbers to the nearest integer. MySQL includes the following function to allow you to round off numbers:

```
ROUND(<number> [, <decimal>])
```

To round off a number, you must specify that number as an argument of the function. Optionally, you can round off a number to a fractional value by specifying the number of decimal places that you want the returned value to include. For example, the following `SELECT` statement rounds off a number to two decimal places:

```
SELECT ROUND(4.27943, 2);
```

In this case, the `ROUND()` function rounds off 4.27943 to 4.28. As you can see, the number is rounded up. Different implementations of the C library, however, might round off numbers in different ways. For example, some might always round numbers up or always down. If you want to have more control over how a number is rounded, you can use the `FLOOR()` or `CEILING()` functions, or you can use the `TRUNCATE()` function, which is shown in the following syntax:

```
TRUNCATE(<number>, <decimal>)
```

The `TRUNCATE()` function takes the same arguments as the `ROUND()` function, except that `<decimal>` is not optional in `TRUNCATE()` functions. You can declare the decimal value as zero, which has the same effect as not including the argument. The main functional difference between `TRUNCATE()` and `ROUND()` is that `TRUNCATE()` always rounds a number toward zero. For example, suppose you modify the last example `SELECT` statement to use `TRUNCATE()`, rather than `ROUND()`, as shown in the following example:

```
SELECT TRUNCATE(4.27943, 2);
```

This time, the value 4.27 is returned, rather than 4.28, because the original value is rounded toward zero, which means, for positive numbers, it is rounded down, rather than up.

SQRT() Function

The `SQRT()` function returns to the square root of a specified number:

```
SQRT(<number>)
```

To use the function, you need to specify the original number as an argument of the function. For example, the following `SELECT` statement uses the `SQRT()` function to find the square root of 36:

```
SELECT SQRT(36);
```

The statement returns a value of 6.

The following exercise allows you to try out several of the numeric functions that you learned about in this section. Because the DVDRentals database doesn't include any tables that are useful to test out these functions, you first must create a table named Test and then add values to that table. From there, you can create `SELECT` statements that use numeric functions to calculate data in the Test table.

Try It Out Using Numeric Functions in Your SQL Statements

Follow these steps to create and populate the Test table and then use `SELECT` statements that employ numeric functions:

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. First, you create the Test table. Execute the following SQL statement at the mysql command prompt:

```
CREATE TABLE Test
(
   TestID SMALLINT NOT NULL PRIMARY KEY,
   Amount SMALLINT NOT NULL
);
```

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

3. Next, add values to the Test table. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO Test
VALUES (101, 12), (102, 1), (103, 139), (104, -37), (105, 0), (106, -16);
```

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

4. The first `SELECT` statement that you create includes the `COT()` function, which calculates the cotangent of the values in the Amount column. Execute the following SQL statement at the mysql command prompt:

```
SELECT TestID, Amount, COT(Amount) AS Cotangent
FROM Test
ORDER BY TestID;
```

   You should receive results similar to the following:

```
+--------+--------+------------------+
| TestID | Amount | Cotangent        |
+--------+--------+------------------+
|    101 |     12 | -1.5726734063977 |
|    102 |      1 | 0.64209261593433 |
|    103 |    139 |  1.0314388663087 |
|    104 |    -37 |  1.1893841441106 |
|    105 |      0 |             NULL |
|    106 |    -16 | -3.3263231956354 |
+--------+--------+------------------+
6 rows in set (0.00 sec)
```

5. The next statement that you create builds on the previous statement by adding a column to the result set that rounds off the cotangent. Execute the following SQL statement at the mysql command prompt:

```
SELECT TestID, Amount, COT(Amount) AS Cotangent, ROUND(COT(Amount)) AS Rounded
FROM Test
ORDER BY TestID;
```

You should receive results similar to the following:

```
+--------+--------+------------------+---------+
| TestID | Amount | Cotangent        | Rounded |
+--------+--------+------------------+---------+
|    101 |     12 | -1.5726734063977 |      -2 |
|    102 |      1 | 0.64209261593433 |       1 |
|    103 |    139 |  1.0314388663087 |       1 |
|    104 |    -37 |  1.1893841441106 |       1 |
|    105 |      0 |             NULL |    NULL |
|    106 |    -16 | -3.3263231956354 |      -3 |
+--------+--------+------------------+---------+
6 rows in set (0.00 sec)
```

6. In the next `SELECT` statement, you use the `MOD()` function to divide the Amount values by 10 and then return the remainder from that division. Execute the following SQL statement at the mysql command prompt:

```
SELECT TestID, Amount, MOD(Amount, 10) AS Modulo
FROM Test
ORDER BY TestID;
```

You should receive results similar to the following:

```
+--------+--------+--------+
| TestID | Amount | Modulo |
+--------+--------+--------+
|    101 |     12 |      2 |
|    102 |      1 |      1 |
|    103 |    139 |      9 |
|    104 |    -37 |     -7 |
|    105 |      0 |      0 |
|    106 |    -16 |     -6 |
+--------+--------+--------+
6 rows in set (0.00 sec)
```

7. Next you create a `SELECT` statement that uses the `POW()` function to raise the Amount values by a power of 2. Execute the following SQL statement at the mysql command prompt:

```
SELECT TestID, Amount, POW(Amount, 2) AS Raised2
FROM Test
ORDER BY TestID;
```

You should receive results similar to the following:

```
+--------+--------+---------+
| TestID | Amount | Raised2 |
+--------+--------+---------+
|    101 |     12 |     144 |
|    102 |      1 |       1 |
|    103 |    139 |   19321 |
|    104 |    -37 |    1369 |
|    105 |      0 |       0 |
|    106 |    -16 |     256 |
+--------+--------+---------+
6 rows in set (0.00 sec)
```

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

```
DROP TABLE Test;
```

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

How It Works

The first SELECT statement that you created includes the COT() function:

```
SELECT TestID, Amount, COT(Amount) AS Cotangent
FROM Test
ORDER BY TestID;
```

The function is used as one of the elements in the select list. The Amount column is specified as the argument used by the COT() function. As a result, for each row returned, the cotangent of the Amount value is calculated and returned in the Cotangent column.

The next SELECT statement that you created is similar to the previous one, except that it includes an additional column in the result set:

```
SELECT TestID, Amount, COT(Amount) AS Cotangent, ROUND(COT(Amount)) AS Rounded
FROM Test
ORDER BY TestID;
```

The Rounded column takes the cotangent for each value and rounds the number to an integer. Notice that the COT() function is embedded as an argument in the ROUND() function. This demonstrates that numeric functions, like other functions, can be embedded to as many layers as necessary.

In the next SELECT statement that you created, you used a MOD() function on the Amount column:

```
SELECT TestID, Amount, MOD(Amount, 10) AS Modulo
FROM Test
ORDER BY TestID;
```

For each row returned by this statement, the Amount value is divided by 10. The remainder from this value is then returned in the Modulo column.

The final statement that you created is similar to the last statement except that it uses the `POW()` function, rather than the `MOD()` function:

```
SELECT TestID, Amount, POW(Amount, 2) AS Raised2
FROM Test
ORDER BY TestID;
```

In this case, for each row returned by the `SELECT` statement, the value in the Amount column is raised by a power of 2 and then returned in the Rasied2 column.

Date/Time Functions

The next set of functions covered are those related to date and time values. These functions are handy for comparing and calculating dates and times as well as returning the current dates and times. MySQL supports numerous date/time functions, and this section covers many of those.

ADDDATE(), DATE_ADD(), SUBDATE(), DATE_SUB(), and EXTRACT() Functions

The `ADDDATE()` and `DATE_ADD()` functions, which are synonymous, allow you to add date-related intervals to your date values, as shown in the following syntax:

```
ADDDATE(<date>, INTERVAL <expression> <type>)
```

As you can see from the syntax, the function includes two arguments, the `<date>` value and the `INTERVAL` clause. The `<date>` value can be any date or date/time literal value or value derived from an expression. This value acts as the root value to which time is added. The `INTERVAL` clause requires an `<expression>`, which must be a time value in an acceptable format, and a `<type>` value. The following table lists the types that you can specify in the `INTERVAL` clause and the format for the expression used with that type:

<type>	**<expression> format**
MICROSECOND	<microseconds>
SECOND	<seconds>
MINUTE	<minutes>
HOUR	<hours>
DAY	<days>
MONTH	<months>
YEAR	<years>
SECOND_MICROSECOND	'<seconds>.<microseconds>'
MINUTE_MICROSECOND	'<minutes>.<microseconds>'
MINUTE_SECOND	'<minutes>:<seconds>'

Table continued on following page

<type>	<expression> format
HOUR_MICROSECOND	'<hours>.<microseconds>'
HOUR_SECOND	'<hours>:<minutes>:<seconds>'
HOUR_MINUTE	'<hours>:<minutes>'
DAY_MICROSECOND	'<days>.<microseconds>'
DAY_SECOND	'<days> <hours>:<minutes>:<seconds>'
DAY_MINUTE	'<days> <hours>:<minutes>'
DAY_HOUR	'<days> <hours>'
YEAR_MONTH	'<years>-<months>'

The best way to understand how the types and expression formats work is to look at an example. The following `SELECT` statement uses the `ADDDATE()` function to add 10 hours and 20 minutes to the specified date/time value:

```
SELECT ADDDATE('2004-10-31 13:39:59', INTERVAL '10:20' HOUR_MINUTE);
```

As you can see, the first argument in the `ADDDATE()` function is the base date/time value, and the second argument is the `INTERVAL` clause. In this clause, the expression used (`'10:20'`) is consistent with the type used (`HOUR_MINUTE`). If you refer back to the table, notice that the expression is in the format acceptable for this type. As a result, the value returned by this statement is `2004-10-31 23:59:59`, which is 10 hours and 20 minutes later than the original date/time value.

The `ADDDATE()` function also includes a second form, which is shown in the following syntax:

```
ADDDATE(<date>, <days>)
```

This form of the `ADDDATE()` syntax allows you to specify a date value as the first argument and a number of days as the second argument. These are the number of days that are to be added to the specified date. For example, the following `SELECT` statement adds 31 days to the date in the first argument:

```
SELECT ADDDATE('2004-11-30 23:59:59', 31);
```

The statement returns a result of `2004-12-31 23:59:59`, which is 31 days after the original date. Notice that the time value remains the same.

In addition to being able to add to a date, you can also subtract from a date by using the `SUBDATE()` or `DATE_SUB()` functions, which are synonymous, as shown in the following syntax:

```
SUBDATE(<date>, INTERVAL <expression> <type>)
```

The arguments used in this syntax are the same as those used for the `ADDDATE()` syntax. For example, the following statement subtracts 12 hours and 10 minutes from the specified date:

```
SELECT SUBDATE('2004-10-31 23:59:59', INTERVAL '12:10' HOUR_MINUTE);
```

The statement returns a value of 2004-10-31 11:49:59, which is 12 hours and10 minutes earlier than the original date.

The SUBDATE() function also includes a second form, which allows you to subtract a specified number of days from a date:

```
SUBDATE(<date>, <days>)
```

For example, the following SELECT statement subtracts 31 days from the specified date:

```
SELECT SUBDATE('2004-12-31 23:59:59', 31);
```

The value returned by this statement is 2004-11-30 23:59:59, 31 days earlier than the original date.

Functions like ADDDATE() and SUBDATE() are useful when you need to change a date value stored in a database but you don't know the new value, only the interval change of that value. For example, suppose that you are building an application for a company that rents computers. You want the application to include a way for users to be able to add days to the date that the equipment must be returned. For example, suppose that the equipment is due back on November 8, but you want to add three days to that date so that the due date is changed to November 11. You can use the ADDDATE() function along with the DAY interval type and set up the application to allow users to enter the number of days. The value returned by the ADDDATE() function can then be inserted in the appropriate column.

Another useful time-related function is the EXTRACT() function, which is shown in the following syntax:

```
EXTRACT(<type> FROM <date>)
```

The function uses the same <type> values that are used for the ADDDATE(), DATE_ADD(), SUBDATE(), and DATE_SUB() functions. In this case, the type extracts a specific part of the date/time value. In addition, when using an EXTRACT() function, you must specify the FROM keyword and a date value. For example, the following SELECT statement extracts the year and month from the specified date:

```
SELECT EXTRACT(YEAR_MONTH FROM '2004-12-31 23:59:59');
```

The EXTRACT() function in this statement includes the type YEAR_MONTH. As a result, the year (2004) and month (12) are extracted from the original value and returned as 200412.

The EXTRACT() function is handy if you have an application that must display only part of a date. For example, suppose that a table in your database includes a TIMESTAMP column. Your application should display only the date portion of the column value. You can use the EXTRACT() function to retrieve only the date portion of the value and use that date portion in your application.

CURDATE(), CURRENT_DATE(), CURTIME(), CURRENT_TIME(), CURRENT_TIMESTAMP(), and NOW() Functions

MySQL includes a number of functions that allow you to retrieve current date and time information. The first of these are the CURDATE() and CURRENT_DATE() functions, which are synonymous. As the following syntax shows, the functions do not require any arguments:

```
CURDATE()
```

To use this function, you simply specify the function in your statement. For example, if you want to retrieve the current date, you can use the following SELECT statement:

```
SELECT CURDATE();
```

The statement retrieves a value similar to 2004-09-08. Notice that the value includes first the year, then the month, and then the day.

You can retrieve the current time by using the CURTIME() or CURRENT_TIME functions, which are also synonymous:

```
CURTIME()
```

Again, you do not need to supply any arguments, as shown in the following SELECT statement:

```
SELECT CURTIME();
```

As you can see, you simply use the function as is to retrieve the information. The date returned is in the same format as the following value: 16:07:46. The time value is listed by hour, then by minute, and then by second.

In addition to retrieving only the date or only the time, you can retrieve both in one value by using the NOW() or CURRENT_TIMESTAMP() functions, which are also synonymous:

```
NOW()
```

The function is used just like CURDATE() or CURTIME(), as shown in the following SELECT statement:

```
SELECT NOW();
```

The value returned by this function is in the same format as the following value: 2004-09-08 16:08:00. The value contains two parts, first the date and then the time. The date value includes first the year, then the month, and then the day. The time value includes first the hour, then the minute, and then the second.

The CURDATE(), CURTIME(), and NOW() functions are particularly useful if you need to insert a date in a column that is based on the current date or time. For example, suppose that your application includes a table that tracks when a user has checked a book out of the library and when that book is due back. You can use the CURDATE() function to determine the current date and insert that function in the column that tracks today's date. You can then use the same function to calculate when the book is due back by adding the correct number of days to today's date. For example, if the book is due back 21 days from today, you can add 21 to CURDATE() to return a date 21 days from today.

DATE(), MONTH(), MONTHNAME(), and YEAR() Functions

MySQL also includes a set of functions that allow you to extract specific information from a date or time value. For example, you can use the following function to extract just the date:

```
DATE(<date>)
```

In this function, the `<date>` value usually represents a date/time value from which you want to extract only the date, as in the following statement:

```
SELECT DATE('2004-12-31 23:59:59');
```

This statement retrieves the full date value of 2004-12-31. You can extract only the month by using the `MONTH()` function:

```
MONTH(<date>)
```

If you were to update the last `SELECT` statement to include `MONTH()` rather than `DATE()`, the statement would be as follows:

```
SELECT MONTH('2004-12-31 23:59:59');
```

This `SELECT` statement retrieves only the month number, which is 12. If you prefer to retrieve the actual month name, you would use the following function:

```
MONTHNAME(<date>)
```

As you can see, you simply use the keyword `MONTHNAME` rather than `MONTH`, as shown in the following `SELECT` statement:

```
SELECT MONTHNAME('2004-12-31 23:59:59');
```

Now your result is the value December, instead of 12.

You can also extract the year from a date value by using the `YEAR()` function:

```
YEAR(<date>)
```

The function returns the year from any date or date/time value, as the following `SELECT` statement demonstrates:

```
SELECT YEAR('2004-12-31 23:59:59');
```

The statement returns the value 2004.

The `DATE()`, `MONTH()`, `MONTHNAME()`, and `YEAR()` functions are helpful when you want to retrieve a portion of a date or a related value based on the date and use it in your application. For example, suppose that you are developing an application that displays individual orders that your company has processed. For each order processed, you want to display the name of the month that the order was taken. You can use the `MONTHNAME()` function to extract the name of the month from the order date, as it's recorded in the database. That way, you don't have to store all the month names in your database, but they're readily available when you retrieve an order.

DATEDIFF() and TIMEDIFF() Functions

You can also use functions to determine the differences between dates and times. For example, the following function calculates the number of dates that separates two dates:

```
DATEDIFF(<date>, <date>)
```

To use the function, you must specify the dates as arguments. For example, the following SELECT statement specifies two dates that are exactly one year apart:

```
SELECT DATEDIFF('2004-12-31 23:59:59', '2003-12-31 23:59:59');
```

The statement returns a value of 366 (because 2004 is a leap year). Notice that the most recent date is specified first. You can specify them in any order, but if the less recent date is specified first, your results are a negative number because of the way dates are compared.

You can also compare time values by using the TIMEDIFF() function:

```
TIMEDIFF(<time>, <time>)
```

This function works similarly to the way that the DATEDIFF() function works. You must specify two time or date/time values, as shown in the follow SELECT statement:

```
SELECT TIMEDIFF('2004-12-31 23:59:59', '2004-12-30 23:59:59');
```

This time, the time difference is returned as 24:00:00, indicating that the time difference between the two is exactly 24 hours.

The DATEDIFF() and TIMEDIFF() functions are useful when you have a table that includes two time/date columns. For example, suppose that you have a table that tracks project delivery dates. The table includes the date that the project started and the date that the project was completed. You can use the TIMEDIFF() function to calculate the number of days that each project took. You can then use that information in your applications or reports or however you want to use it.

DAY(), DAYOFMONTH(), DAYNAME(), DAYOFWEEK(), and DAYOFYEAR() Functions

MySQL also allows you to pull day-related values out of date or date/time values. For example, the DAY() and DAYOFMONTH() functions, which are synonymous, extract the day of the month out of a value. The DAY() function is shown in the following syntax:

```
DAY(<date>)
```

As you can see, only one argument is required. For example, the following SELECT statement includes a DAY() function with a time/date value as an argument:

```
SELECT DAY('2004-12-31 23:59:59');
```

The day in this case is 31, which is the value returned by this statement. You can also return the name of the day by using the following function:

```
DAYNAME(<date>)
```

You can use the `DAYNAME()` function with any date or date/time value, as shown in the following example:

```
SELECT DAYNAME('2004-12-31 23:59:59');
```

In this example, the function calculates which day is associated with the specified date and returns that day as a value. In this case, the value returned is Friday. If you want to return the day of the week by number, you would use the following function:

```
DAYOFWEEK(<date>)
```

The function returns a value from 1 through 7, with Sunday being 1. For example, the following `SELECT` statement calculates the day of the week for December 31, 2004.

```
SELECT DAYOFWEEK('2004-12-31 23:59:59');
```

The day in this case is Friday. Because Friday is the sixth day, the statement returns a value of 6. In addition, you can calculate a numerical value for the day, as it falls in the year, by using the `DAYOFYEAR()` function:

```
DAYOFYEAR(<date>)
```

In this case, the day is based on the number of days in the year, starting with January 1. For example, the following statement calculates the day of year for the last day of 2004:

```
SELECT DAYOFYEAR('2004-12-31 23:59:59');
```

Because 2004 is a leap year, the statement returns the value 366.

The `DAY()`, `DAYOFMONTH()`, `DAYNAME()`, `DAYOFWEEK()`, and `DAYOFYEAR()` functions can be useful if you want to extract specific types of information from a date. For example, suppose that you are developing an application that tracks employee sick days. When the application pulls dates from the database, it should also display the day of the week that the employee was sick. You can use the `DAYOFWEEK()` function when retrieving the date value from the database to display the day of the week in your application.

SECOND(), MINUTE(), HOUR(), and TIME() Functions

You can also use functions to extract time parts from a time or date/time value. For example, the following function extracts the seconds from a time value:

```
SECOND(<time>)
```

As you would expect, the `SECOND()` function determines the seconds based on the value specified as an argument in the function. For example, the following `SELECT` statement extracts 59 from the specified value.

```
SELECT SECOND('2004-12-31 23:59:59');
```

You can also extract the minutes by using the `MINUTE()` function:

```
MINUTE(<time>)
```

If you were to modify the last statement to use the `MINUTE()` function, it would extract the minutes from the specified date. For example, the following statement would extract 59 from the time value:

```
SELECT MINUTE('2004-12-31 23:59:59');
```

The following function allows you to extract the hour from the time value:

```
HOUR(<time>)
```

As shown in the following `SELECT` statement, the `HOUR()` function extracts 23 from the time value:

```
SELECT HOUR('2004-12-31 23:59:59');
```

You can also extract the entire time value by using the `TIME()` function:

```
TIME(<time>)
```

Using this function returns the hour, minutes, and seconds. For example, the following `SELECT` statement includes a date/time value as an argument in the `TIME()` function:

```
SELECT TIME('2004-12-31 23:59:59');
```

In this case, the function returns the value 23:59:59.

The `SECOND()`, `MINUTE()`, `HOUR()`, and `TIME()` functions are similar to the `DAY()`, `DAYOFMONTH()`, `DAYNAME()`, `DAYOFWEEK()`, and `DAYOFYEAR()` functions in how they can be used in your applications. For example, suppose that you are creating an application to track orders. The order data is stored in a table in your database. Each time an order is added to the table, a value is automatically inserted in the `TIMESTAMP` column. For tracking purposes, you want your application to display the hour that each order was taken as a field separate from the rest of the `TIMESTAMP` value. You can use the `TIME()` function to extract only the time from the `TIMESTAMP` value and use that time value in your application.

Now that you've seen how to use many of the date/time functions in MySQL, you're ready to try some out. This exercise uses a number of `SELECT` statements to retrieve date- and time-related data from the DVDRentals database as well as to retrieve current date and time date.

Try It Out Using Date/Time Functions in Your SQL Statements

The following steps describe how to create statements that include date/time functions:

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

```
use DVDRentals
```

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

2. The first SELECT statement that you create uses the DATEDIFF() function to determine the difference between DateOut and DateDue values in the Transactions table. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDID, DateOut, DateDue, DATEDIFF(DateDue, DateOut) AS TotalDays
FROM Transactions
WHERE TransID=11;
```

You should receive results similar to the following:

```
+-------+------------+------------+-----------+
| DVDID | DateOut    | DateDue    | TotalDays |
+-------+------------+------------+-----------+
|     1 | 2004-09-06 | 2004-09-09 |         3 |
+-------+------------+------------+-----------+
1 row in set (0.00 sec)
```

3. Next you use the DAYNAME() function to retrieve the name of a day for a specific date. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDID, DateDue, DAYNAME(DateDue) AS DayDue
FROM Transactions
WHERE TransID=11;
```

You should receive results similar to the following:

```
+-------+------------+----------+
| DVDID | DateDue    | DayDue   |
+-------+------------+----------+
|     1 | 2004-09-09 | Thursday |
+-------+------------+----------+
1 row in set (0.00 sec)
```

4. In the next SELECT statement, you use the YEAR() function to extract the year from a date value. Execute the following SQL statement at the mysql command prompt:

```
SELECT TransID, YEAR(DateOut) AS YearOut
FROM Transactions
WHERE TransID>15
ORDER BY TransID;
```

You should receive results similar to the following:

```
+---------+---------+
| TransID | YearOut |
+---------+---------+
|      16 |    2004 |
|      17 |    2004 |
|      18 |    2004 |
|      19 |    2004 |
|      20 |    2004 |
|      21 |    2004 |
|      22 |    2004 |
|      23 |    2004 |
+---------+---------+
8 rows in set (0.00 sec)
```

5. Next, use the `DATE_ADD()` function to add four days to a DateDue value in the Transactions table. Execute the following SQL statement at the mysql command prompt:

```
SELECT TransID, DateDue, DATE_ADD(DateDue, INTERVAL 4 DAY) AS Add4Days
FROM Transactions
WHERE TransID=11;
```

You should receive results similar to the following:

```
+---------+------------+------------+
| TransID | DateDue    | Add4Days   |
+---------+------------+------------+
|      11 | 2004-09-09 | 2004-09-13 |
+---------+------------+------------+
1 row in set (0.00 sec)
```

6. Your last `SELECT` statement retrieves the current date and time from your system. Execute the following SQL statement at the mysql command prompt:

```
SELECT CURDATE(), CURTIME(), NOW();
```

You should receive results similar to the following:

```
+------------+-----------+---------------------+
| CURDATE()  | CURTIME() | NOW()               |
+------------+-----------+---------------------+
| 2004-09-08 | 16:26:42  | 2004-09-08 16:26:42 |
+------------+-----------+---------------------+
1 row in set (0.00 sec)
```

How It Works

The first `SELECT` statement that you created includes the `DATEDIFF()` function as an element in the select list:

```
SELECT DVDID, DateOut, DateDue, DATEDIFF(DateDue, DateOut) AS TotalDays
FROM Transactions
WHERE TransID=11;
```

The function uses the values in the DateDue and DateOut columns of the Transactions table to provide date values as arguments in the function. The difference in days is then displayed in the TotalDays column of the result set.

In the next `SELECT` statement, you used the `DAYNAME()` function to retrieve the name of the day from a date in the DateDue column of the Transactions table:

```
SELECT DVDID, DateDue, DAYNAME(DateDue) AS DayDue
FROM Transactions
WHERE TransID=11;
```

The `DAYNAME()` function is used as an argument in the select list to return the name of the day to the DayDue column of the result set.

Next, you created a SELECT statement that includes a YEAR() function as an element in the select list:

```
SELECT TransID, YEAR(DateOut) AS YearOut
FROM Transactions
WHERE TransID>15
ORDER BY TransID;
```

The YEAR() function extracts the year from the DateOut values of the Transactions table and displays those years in the YearOut column of the result set.

In the next SELECT statement, you used the DATE_ADD() function to add four days to the value in the DateDue column:

```
SELECT TransID, DateDue, DATE_ADD(DateDue, INTERVAL 4 DAY) AS Add4Days
FROM Transactions
WHERE TransID=11;
```

As with the other functions used in this exercise, the DATE_ADD() function is an element in the select list. The function's first argument is the base date value, which is pulled from the DateDue column of the Transactions table. The function's second argument is the INTERVAL clause, which specifies that four days should be added to the date. The new day is then displayed in the Add4Days columns of the result set.

Finally, the last SELECT statement that you created retrieved current date and time information from your system:

```
SELECT CURDATE(), CURTIME(), NOW();
```

The statement includes three functions: CURDATE(), CURTIME(), and NOW(). Each function is an element in the select list. The CURDATE() function returns the current date, the CURTIME() function returns the current time, and the NOW() function returns both.

Summarizing Data

In Chapter 7, you learned how you can add the GROUP BY clause to a SELECT statement in order to summarize data. To provide effective summarizing capabilities, MySQL includes a number of functions — referred to as *aggregate* functions — that calculate or compare values based on the columns specified in the GROUP BY clause. In this section, you learn about many of the aggregate functions and how they're used in a SELECT statement. The examples in this section are based on the following table definition:

```
CREATE TABLE Classes
(
   ClassID SMALLINT NOT NULL PRIMARY KEY,
   Dept CHAR(4) NOT NULL,
   Level ENUM('Upper', 'Lower') NOT NULL,
   TotalStudents TINYINT UNSIGNED NOT NULL
);
```

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

```
INSERT INTO Classes
VALUES (1001, 'ANTH', 'Upper', 25),
(1002, 'ANTH', 'Upper', 25),
(1003, 'MATH', 'Upper', 18),
(1004, 'ANTH', 'Lower', 19),
(1005, 'ENGL', 'Upper', 28),
(1006, 'MATH', 'Lower', 23),
(1007, 'ENGL', 'Upper', 25),
(1008, 'MATH', 'Lower', 29),
(1009, 'ANTH', 'Upper', 25),
(1010, 'ANTH', 'Lower', 30),
(1011, 'ENGL', 'Lower', 26),
(1012, 'MATH', 'Lower', 22),
(1013, 'ANTH', 'Upper', 27),
(1014, 'ANTH', 'Upper', 21),
(1015, 'ENGL', 'Lower', 25),
(1016, 'ENGL', 'Upper', 32);
```

This section uses the Classes table to demonstrate how to add aggregate functions to your SELECT statements. For the purposes of this discussion, the functions have been divided into two broad categories: basic summary functions and bit functions.

Summary Functions

Summary functions are the basic functions that you're most likely to use when using a GROUP BY clause to group together columns of data. The summary functions allow you to perform such tasks as determining the average of a group of values or adding those values together.

AVG() Function

The first aggregate function that you learn about is the AVG() function, which averages the values returned by a specified expression. The following syntax shows how to use the function:

```
AVG(<expression>)
```

Nearly all aggregate functions use similar syntax. Each one requires that you specify some type of expression. In most cases, the expression is simply the name of a column that is significant in terms of how data is being grouped together in the GROUP BY clause. For example, the following SELECT statement uses the AVG() function to average the values in the TotalStudents column:

```
SELECT Dept, Level, ROUND(AVG(TotalStudents)) AS Average
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

As you can see, the statement includes a GROUP BY clause that specifies that the results be grouped together first by the Dept column and then by the Level column. The AVG() function then calculates the values in the TotalStudents column. The TotalStudents column is the only column significant to the way that the data is being grouped together. Finding the average of the ClassID column (the only other column in the Classes table) would have no meaning in this sense; however, the number of students is meaningful to the groups. As a result, the values returned by the SELECT statement include an average number of students for each department at each level, as shown in the following result set:

```
+------+-------+---------+
| Dept | Level | Average |
+------+-------+---------+
| ANTH | Upper |      25 |
| ANTH | Lower |      24 |
| ANTH | NULL  |      25 |
| ENGL | Upper |      28 |
| ENGL | Lower |      25 |
| ENGL | NULL  |      27 |
| MATH | Upper |      18 |
| MATH | Lower |      25 |
| MATH | NULL  |      23 |
| NULL | NULL  |      25 |
+------+-------+---------+
10 rows in set (0.00 sec)
```

As you can see, an average is provided for each group. For example, the average class size for an Upper level ANTH class is 25, but the average class size for a Lower level ANTH class is 24.

One thing you might have also noticed is that the statement uses the ROUND() function to round the values returned by the AVG() function. If you had not used the ROUND() function, the values in the Average column would have been fractional. Another aspect of the statement that you might have noticed is the use of the WITH ROLLUP option in the GROUP BY clause. As a result, averages are provided for the departments as a whole and for all classes added together. For example, the ANTH department has an average class size of 25 students per class, while the ENGL department has an average class size of 27 students.

SUM() Function

The next aggregate function is the SUM() function, which is shown in the following syntax:

```
SUM(<expression>)
```

The SUM() function returns the sum of the expression. In most cases, this means that the values in a column are added together, based on how columns are grouped together in the GROUP BY clause. For example, suppose you modify the last SELECT statement to include the SUM() function rather than the AVG() and ROUND() functions:

```
SELECT Dept, Level, SUM(TotalStudents) AS Total
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

By using the `SUM()` function, the values in the TotalStudents column are added together, as shown in the following result set:

```
+------+-------+-------+
| Dept | Level | Total |
+------+-------+-------+
| ANTH | Upper |   123 |
| ANTH | Lower |    49 |
| ANTH | NULL  |   172 |
| ENGL | Upper |    85 |
| ENGL | Lower |    51 |
| ENGL | NULL  |   136 |
| MATH | Upper |    18 |
| MATH | Lower |    74 |
| MATH | NULL  |    92 |
| NULL | NULL  |   400 |
+------+-------+-------+
10 rows in set (0.00 sec)
```

Now each grouping includes the total number of students. For example, the total number of students that attend the Upper level ANTH classes is 123, compared to 49 for the Lower level.

MIN() and MAX() Functions

In addition to calculating averages and sums, you can also find the minimum or maximum value in a column. For example, the following aggregate function returns the minimum value from a group of values:

```
MIN(<expression>)
```

If you modify the last `SELECT` statement to use `MIN()` instead of `SUM()`, your statement reads as follows:

```
SELECT Dept, Level, MIN(TotalStudents) AS Minimum
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

Notice that nothing has changed in the statement except the function keyword and the name assigned to the column. Now the results include the minimum value for each group, as shown in the following result set:

```
+------+-------+---------+
| Dept | Level | Minimum |
+------+-------+---------+
| ANTH | Upper |      21 |
| ANTH | Lower |      19 |
| ANTH | NULL  |      19 |
| ENGL | Upper |      25 |
| ENGL | Lower |      25 |
| ENGL | NULL  |      25 |
| MATH | Upper |      18 |
| MATH | Lower |      22 |
| MATH | NULL  |      18 |
| NULL | NULL  |      18 |
+------+-------+---------+
10 rows in set (0.00 sec)
```

As the results show, the minimum number of students in an Upper level anthropology class is 21. You can also calculate the maximum value by using the MAX() function:

```
MAX(<expression>)
```

You can then modify your SELECT statement as follows:

```
SELECT Dept, Level, MAX(TotalStudents) AS Maximum
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

Now your result set includes the maximum number of students in each type of class:

```
+------+-------+---------+
| Dept | Level | Maximum |
+------+-------+---------+
| ANTH | Upper |      27 |
| ANTH | Lower |      30 |
| ANTH | NULL  |      30 |
| ENGL | Upper |      32 |
| ENGL | Lower |      26 |
| ENGL | NULL  |      32 |
| MATH | Upper |      18 |
| MATH | Lower |      29 |
| MATH | NULL  |      29 |
| NULL | NULL  |      32 |
+------+-------+---------+
10 rows in set (0.00 sec)
```

As you can see, the largest ANTH class size is in the Lower level, with 30 students in that class.

COUNT() Function

Another very useful aggregate function is COUNT(), which is shown in the following syntax:

```
COUNT([DISTINCT] {<expression> | *})
```

The COUNT() function is the only aggregate function that deviates from the basic syntax used by the other aggregate functions. As you can see, there are two differences. You can include the DISTINCT keyword, and you can specify an asterisk (*) rather than an expression. You should use the DISTINCT keyword if you want to eliminate duplicates from the count. The asterisk is useful if you want to count all the rows, whether or not they contain NULL values. If you specify a column rather than an asterisk, and that column contains NULL values, the rows that contain the NULL values are not included in the count. If the column contains no NULL values, then the results are the same whether you specify a column name or an asterisk.

Return to the example SELECT statement and replace the MAX() function with the COUNT() function:

```
SELECT Dept, Level, COUNT(*) AS NumberClasses
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

Notice that the statement uses the asterisk rather than specifying a column name. By using the `COUNT()` function, the statement now returns the number of classes in each group, as shown in the following result set:

```
+------+-------+---------------+
| Dept | Level | NumberClasses |
+------+-------+---------------+
| ANTH | Upper |             5 |
| ANTH | Lower |             2 |
| ANTH | NULL  |             7 |
| ENGL | Upper |             3 |
| ENGL | Lower |             2 |
| ENGL | NULL  |             5 |
| MATH | Upper |             1 |
| MATH | Lower |             3 |
| MATH | NULL  |             4 |
| NULL | NULL  |            16 |
+------+-------+---------------+
10 rows in set (0.01 sec)
```

As you can see, the `COUNT()` function adds together the number of rows in each group. For example, the number of rows in the ANTH/Upper group is 5. Because each row in the table represents one class, the ANTH/Upper group contains five classes.

Bit Functions

The bit aggregate functions work in the same way that bit operators work. (Refer to Chapter 8 for more information about bit operators.) When values in a grouped column are calculated using a bit function, the bits are compared and a value is returned. The way in which the bits are compared depends on the bit function being used.

The first bit aggregate function is the `BIT_AND()` function. The function uses the following syntax:

```
BIT_AND(<expression>)
```

The `BIT_AND()` function is similar to the bitwise `AND` (`&`) comparison operator in the way in which the function compares bits. For any two or more values, the bits in each bit position are compared. Based on the comparison, a 1 is returned if all bits in a bit position equal 1; otherwise, a 0 is returned. For example, suppose that the Classes table that you have been using in this section includes an additional column that tracks attributes that describe the class, such as room requirements or special equipment. You are developing an application that uses the Attributes column to determine class-specific needs. The following `SELECT` statement uses the `BIT_AND()` function to compare bits in the Attributes column:

```
SELECT Dept, Level, BIT_AND(Attributes) AS BitwiseAND
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

As the statement shows, the `BIT_AND()` function takes the TotalStudents column as an argument. Your application can then use this information to form a bigger picture of the class attributes. For example, these results would tell you if every class is going to need special equipment of some type. As a result, the bits for each value in this column are compared, and the following results are returned:

```
+------+-------+------------+
| Dept | Level | BitwiseAND |
+------+-------+------------+
| ANTH | Upper |         17 |
| ANTH | Lower |         18 |
| ANTH | NULL  |         16 |
| ENGL | Upper |          0 |
| ENGL | Lower |         24 |
| ENGL | NULL  |          0 |
| MATH | Upper |         18 |
| MATH | Lower |         20 |
| MATH | NULL  |         16 |
| NULL | NULL  |          0 |
+------+-------+------------+
10 rows in set (0.00 sec)
```

The next bit aggregate function is the `BIT_OR()` function, which is shown in the following syntax:

```
BIT_OR(<expression>)
```

This function is similar to the bitwise `OR` (|) bit operator. The function compares bits in each bit position and returns 1 if at least one bit is 1; otherwise, 0 is returned. You can use the `BIT_OR()` function as you use the `BIT_AND()` function:

```
SELECT Dept, Level, BIT_OR(TotalStudents) AS BitwiseOR
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

By using the `BIT_OR()` function, your results are now similar to the following:

```
+------+-------+-----------+
| Dept | Level | BitwiseOR |
+------+-------+-----------+
| ANTH | Upper |        31 |
| ANTH | Lower |        31 |
| ANTH | NULL  |        31 |
| ENGL | Upper |        61 |
| ENGL | Lower |        27 |
| ENGL | NULL  |        63 |
| MATH | Upper |        18 |
| MATH | Lower |        31 |
| MATH | NULL  |        31 |
| NULL | NULL  |        63 |
+------+-------+-----------+
10 rows in set (0.01 sec)
```

As with bit operators, bit functions also provide an exclusive `OR` (`XOR`) function, as shown in the following syntax:

```
BIT_XOR(<expression>)
```

This function is similar to the bitwise exclusive `XOR` (^) bit operator in that it compares bits and returns 1 only if exactly one of the bit equals 1; otherwise, 0 is returned. For example, suppose that you update your `SELECT` statement to include the `BIT_XOR()` function:

```
SELECT Dept, Level, BIT_XOR(TotalStudents) AS BitwiseXOR
FROM Classes
GROUP BY Dept, Level WITH ROLLUP;
```

Now your results are different from either the `BIT_AND()` function or the `BIT_OR()` function, as shown in the following result set:

```
+------+-------+------------+
| Dept | Level | BitwiseXOR |
+------+-------+------------+
| ANTH | Upper |         23 |
| ANTH | Lower |         13 |
| ANTH | NULL  |         26 |
| ENGL | Upper |         37 |
| ENGL | Lower |          3 |
| ENGL | NULL  |         38 |
| MATH | Upper |         18 |
| MATH | Lower |         28 |
| MATH | NULL  |         14 |
| NULL | NULL  |         50 |
+------+-------+------------+
10 rows in set (0.00 sec)
```

As illustrated in this section, aggregate functions are a useful way to summarize data that's been grouped together. In the following example, you try out a couple of these functions by creating `SELECT` statements that group together data in the DVDRentals database.

Try It Out Using Aggregate Functions to Group Data

The following steps describe how to use aggregate functions in your `SELECT` statements:

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

```
use DVDRentals
```

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

2. First, use the `MAX()` function to return the maximum value in the NumDisks column of the DVDs table. Execute the following SQL statement at the mysql command prompt:

```
SELECT MTypeID, MAX(NumDisks) AS MaxDisks
FROM DVDs
GROUP BY MTypeID;
```

You should receive results similar to the following:

```
+---------+----------+
| MTypeID | MaxDisks |
+---------+----------+
| mt11    |        1 |
| mt12    |        2 |
| mt16    |        1 |
+---------+----------+
3 rows in set (0.00 sec)
```

3. Now use the `COUNT()` function to determine the number of rows in each RatingID/MTypeID group in the DVDs table. Execute the following SQL statement at the mysql command prompt:

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

You should receive results similar to the following:

```
+----------+---------+-------+
| RatingID | MTypeID | Total |
+----------+---------+-------+
| G        | mt12    |     1 |
| G        | NULL    |     1 |
| NR       | mt11    |     2 |
| NR       | mt12    |     1 |
| NR       | mt16    |     1 |
| NR       | NULL    |     4 |
| PG       | mt11    |     2 |
| PG       | NULL    |     2 |
| R        | mt12    |     1 |
| R        | NULL    |     1 |
| NULL     | NULL    |     8 |
+----------+---------+-------+
11 rows in set (0.01 sec)
```

How It Works

The first aggregate function that you used in this exercise is the `MAX()` function, as shown in the following statement:

```
SELECT MTypeID, MAX(NumDisks) AS MaxDisks
FROM DVDs
GROUP BY MTypeID;
```

In this statement, you grouped together data based on the values in the MTypeID column. You then used the `MAX()` function to retrieve the maximum value from the NumDisks column in each group. For example, one of the groups of data is based on the MTypeID value of mt12. For this group, the maximum value in the NumDisks column is 2, so that is the value the `MAX()` function returns.

In the next `SELECT` statement that you created, you grouped data together first by the RatingID column and then by the MTypeID column:

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

You then used the `COUNT()` function to return the number of rows in each group. For example, one group is made up of the NR value in the RatingID column and the mt11 value in the MTypeID column. This group is made up of two rows, so the `COUNT()` functions returns a value of 2. In other words, two rows contain a RatingID value of NR and an MTypeID value of mt11.

Performing System Operations

The functions examined so far in this chapter have been specific to the data itself; however, a number of functions are related to system operations. For example, you can use function to encrypt and decrypt data, return system-related information, or provide details about executed query and insert operations. This section covers many of these functions and provides examples that demonstrate how they are used.

Encryption Functions

The encryption functions are used to encrypt and decrypt data. These functions are useful when securing the data in your database. You can use these functions in conjunction with MySQL security to secure your database. (For more information about database security, see Chapter 14.)

ENCODE() and DECODE() Functions

The first encryption function that you look at is the `ENCODE()` function, which encrypts a specified string. To use the function, you must specify two arguments, the string to be encrypted and a key (password), as shown in the following syntax:

```
ENCODE(<string>, <key>)
```

When you use the `ENCODE()` function to encrypt a string, the string is stored in a column as a binary string that is the same length as the original string value. To better understand how this function works, take a look at an example, which is based on the following table definition:

```
CREATE TABLE UserAccounts
(
   UserID SMALLINT NOT NULL PRIMARY KEY,
   Password VARCHAR(20) NOT NULL
);
```

Suppose that you want to insert a row in the UserAccounts table, but you want to encrypt the Password value. To encrypt the Password value, you can use an `INSERT` statement similar to the following:

```
INSERT INTO UserAccounts
VALUES (101, ENCODE('pw101', 'key101'));
```

In this statement, the `ENCODE()` function defines the second value in the `VALUES` clause. The function contains two arguments. The first (`pw101`) is the string that is to be encrypted. The second (`key101`) is the password that decrypts the string value if necessary. Once the row is inserted, you can use the following `SELECT` statement to view the values in the new row:

```
SELECT * FROM Users;
```

The `SELECT` statement returns something similar to the following results:

```
+--------+----------+
| UserID | Password |
+--------+----------+
|    101 | ¢-*'_    |
+--------+----------+
1 row in set (0.00 sec)
```

The result set shows that the Password value is encrypted. The only way that you can read the actual string value is to use the `DECODE()` function, which is shown in the following syntax:

```
DECODE(<encrypted string>, <key>)
```

As you can see, you must supply the encrypted value as well as the password that you used to encrypt that value. For example, you can use the following `SELECT` statement to display the actual Password value:

```
SELECT UserID, DECODE(Password, 'key101') AS Password
FROM UserAccounts;
```

Notice that the `DECODE()` function serves as the second element in the select list. The first argument in the function is the Password column, and the second argument is the password. The statement returns the following results:

```
+--------+----------+
| UserID | Password |
+--------+----------+
|    101 | pw101    |
+--------+----------+
1 row in set (0.00 sec)
```

As you can see, the original string value is displayed in the result set. The value, however, is still stored in the Users table in its encrypted state.

PASSWORD(), MD5(), SHA(), and SHA1() Functions

MySQL provides several functions that support one-way hashing encryption, as opposed to the `ENCODE()` function, which facilitates two-way encryption. The advantage to one-way encryption over two-way is that a one-way is less likely to be compromised as a result of a key value being discovered. In addition, one-way encryption eliminates the need to track the decryption key.

The first of these one-way hashing encryption functions is the `PASSWORD()` function. The `PASSWORD()` function, which is shown in the following syntax, encrypts a specified string as a 41-byte hash value:

```
PASSWORD(<string>)
```

For this function, you need to provide only one argument: the string value to be encrypted. You can test how this function works by using a `SELECT` statement similar to the following:

```
SELECT PASSWORD('MyPassword');
```

The `PASSWORD()` function encrypts the MyPassword string value and returns the following results:

```
+-------------------------------------------+
| PASSWORD('MyPassword')                    |
+-------------------------------------------+
| *ACE88B25517D0F22E5C3C643944B027D56345D2D |
+-------------------------------------------+
1 row in set (0.00 sec)
```

MySQL uses the PASSWORD() function to encrypt passwords that are stored in the user grant table. These are the passwords that are associated with specific MySQL user accounts. You should use this function if you plan to modify those passwords directly, after a user account has been created. (You learn more about setting passwords in Chapter 14.) If you're planning to build an application that uses a MySQL database, and you store the passwords that are used for that application in the database, you should use the MD5() or SHA() functions, rather than the PASSWORD() function, because the MD5() and SHA() functions conform to widely accepted standards that are supported on multiple platforms. If you use the PASSWORD() function to encrypt the passwords, the platform on which your application is running might not support that particular format, preventing users from using the application.

You can use the MD5() function to create a 128-bit encrypted value that is based on the specified string. You use the MD5() function just like the PASSWORD() function, as shown in the following syntax.

```
MD5(<string>)
```

As with the PASSWORD() function, you can try out the MD5() function by using a SELECT statement similar to the following:

```
SELECT MD5('MyPassword');
```

The following result set shows the value returned by using the MD5() to encrypt the MyPassword string value:

```
+----------------------------------+
| MD5('MyPassword')                |
+----------------------------------+
| 48503dfd58720bd5ff35c102065a52d7 |
+----------------------------------+
1 row in set (0.00 sec)
```

If you want to create a 160-bit encrypted value based on a specified string, you can use the SHA() or SHA1() functions, which are synonymous. The SHA() function is shown in the following syntax:

```
SHA(<string>)
```

You can use the SHA() function just as you use the MD5() function:

```
SELECT SHA('MyPassword');
```

When you use the SHA() function to encrypt the MyPassword string value, the following value is returned:

```
+------------------------------------------+
| SHA('MyPassword')                        |
+------------------------------------------+
| daa1f31819ed4928fd00e986e6bda6dab6b177dc |
+------------------------------------------+
1 row in set (0.00 sec)
```

Now that you have an overview of how encryption functions work, you're ready to learn about system-related functions that provide you with information about the MySQL operation.

System-Related Functions

The system-related functions return information about the current users on the system, the connection ID being used, and the current database. You can also use system-related functions to convert IP addresses to numerical values and numerical values to IP addresses.

CURRENT_USER(), SESSION_USER(), SYSTEM_USER(), and USER() Functions

The `CURRENT_USER()`, `SESSION_USER()`, `SYSTEM_USER()`, and `USER()` functions all return the username and the hostname under which the current session is authenticated. In each case, the functions take no arguments. For example, the syntax for the `USER()` function is as follows:

```
USER()
```

You can use the functions in any `SELECT` statement to return the username and hostname, as shown in the following example:

```
SELECT USER();
```

The value returned by these functions is in the form of *`<username>@<hostname>`*, such as root@localhost.

Note that, under some circumstances, the `CURRENT_USER()` *function can sometimes return a value different from the other three functions. For example, if a client specifies a username, but the client is authenticated anonymously, a difference can occur. For more information about these functions, see the MySQL product documentation.*

CONNECTION_ID(), DATABASE(), and VERSION() Functions

MySQL also provides functions that return information about the current connection, database, and MySQL version. For example, the following function returns the current connection ID:

```
CONNECTION_ID()
```

As you can see, the function takes no arguments. If you invoke this function through a `SELECT` statement such as the following, the value returned is a digit that represents the connection ID:

```
SELECT CONNECTION_ID();
```

You can also retrieve the name of the current database by using the `DATEBASE()` function:

```
DATABASE()
```

Again, you can use the function in an SQL statement to retrieve the name of the database. For example, the following `SELECT` statement returns the name of the current database:

```
SELECT DATABASE();
```

If you are using the mysql client tool but are not working in the context of a specific database, a `NULL` is returned.

MySQL also allows you to check which version of MySQL that you're working on:

```
VERSION()
```

Once again, no arguments are required, as shown in the following `SELECT` statement:

```
SELECT VERSION();
```

The statement returns the current version number.

INET_ATON() and INET_NTOA() Functions

If you want to convert a network address to a numeric value, you can use the following `INET_ATON()` function:

```
INET_ATON(<network address>)
```

The function takes one argument, which is the network address. For example, the following `SELECT` statement returns the numeric value for the address 127.0.0.1:

```
SELECT INET_ATON('127.0.0.1');
```

When you execute the statement, MySQL returns the value of 2130706433. You can also take a numeric value such as this and convert it to an IP address by using the following function:

```
INET_NTOA(<network integer>)
```

This function also takes only one argument, which is the numeric representation of a network address. For example, the following `SELECT` statement converts the value 2130706433 to an IP address:

```
SELECT INET_NTOA(2130706433);
```

The statement returns an IP address of 127.0.0.1.

The `INET_ATON()` and `INET_NTOA()` functions are useful if you want to store IP addresses as numerical values, rather than string values. Storing IP addresses as numerical values requires only 4 bytes of storage per value, whereas storing them as string values requires up to 15 bytes.

As you have seen, you can use system-related functions to return information about users, connections, and databases. Now you can examine functions that can be used in conjunction with your query and insert operations.

Query and Insert Functions

The query and insert functions are used for specific types of queries and inserts. They include two functions: the `FOUND_ROWS()` function and the `LAST_INSERT_ID()` function.

FOUND_ROWS() Function

The `FOUND_ROWS()` function is used in conjunction with the `LIMIT` clause of a `SELECT` statement. Recalling from Chapter 7, the `LIMIT` clause specifies the number of rows to be returned in a result set.

There might be times, though, when you want to know how many rows would have been returned had you not used the LIMIT. The FOUND_ROWS() function allows you to determine the number of the entire result set. The function is shown in the following syntax:

```
FOUND_ROWS()
```

As you can see, the FOUND_ROWS() function requires no arguments. To use the function, the SELECT statement that includes the LIMIT clause must also include the SQL_CALC_FOUND_ROWS option. (The SQL_CALC_FOUND_ROWS option is a SELECT statement option that specifies what the row count of a result set would be if the LIMIT clause were not used.) Once you execute that SELECT statement, you can execute a second SELECT statement that calls the FOUND_ROWS() function, as shown in the following example:

```
SELECT FOUND_ROWS();
```

Executing this statement displays the number of rows that would have been returned by the original SELECT statement, had you not used the LIMIT clause.

LAST_INSERT_ID() Function

The LAST_INSERT_ID() function allows you to retrieve the last value that was inserted in an AUTO_INCREMENT column. For example, suppose you insert a row in a table whose primary key column is configured with the AUTO_INCREMENT option. When you insert that row, a value is automatically added to the primary key value. The value increments by one, based on the highest value that already exists in the column. Once you insert a row, you might want to know the value that was inserted in the primary key column. You can use the following function to retrieve that value:

```
LAST_INSERT_ID()
```

As you can see, the function takes no argument. You simply call the function after your last insert operation, as shown in the following SELECT statement:

```
SELECT LAST_INSERT_ID();
```

The value retrieved by the LAST_INSERT_ID() function is specific to a client connection. This way, you won't retrieve an AUTO_INCREMENT value that another client added. You can retrieve only the last value that you generated, not one that someone else generated.

In the following Try It Out exercise, you use several of the system-related functions that you learned about in this last section. To use the functions, you create a number of SELECT statements that include the functions as elements in the select lists of those statements.

Try It Out Using Functions to Perform System-Related Operations

The following steps describe how to create the SELECT statements that use functions to perform system-related operations:

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. First, create a SELECT statement that retrieves information about the current user. Execute the following SQL statement at the mysql command prompt:

```
SELECT USER();
```

You should receive results similar to the following:

```
+----------------+
| USER()         |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.00 sec)
```

3. Now retrieve the name of the current database. Execute the following SQL statement at the mysql command prompt:

```
SELECT DATABASE();
```

You should receive results similar to the following:

```
+------------+
| DATABASE() |
+------------+
| dvdrentals |
+------------+
1 row in set (0.00 sec)
```

4. Next, create a SELECT statement that returns the connection ID. Execute the following SQL statement at the mysql command prompt:

```
SELECT CONNECTION_ID();
```

You should receive results similar to the following (with a connection ID specific to your connection):

```
+-----------------+
| CONNECTION_ID() |
+-----------------+
|              12 |
+-----------------+
1 row in set (0.01 sec)
```

5. Now try out the FOUND_ROWS() function. Before you do that, you must create a SELECT statement that includes the SQL_CALC_FOUND_ROWS option and the LIMIT clause. Execute the following SQL statement at the mysql command prompt:

```
SELECT SQL_CALC_FOUND_ROWS DVDName
FROM DVDs
WHERE StatID='s2'
ORDER BY DVDName
LIMIT 2;
```

You should receive results similar to the following:

```
+----------+
| DVDName  |
+----------+
| Amadeus  |
| Mash     |
+----------+
2 rows in set (0.00 sec)
```

6. Now you can create a SELECT statement that includes the FOUND_ROWS() function. Execute the following SQL statement at the mysql command prompt:

```
SELECT FOUND_ROWS();
```

You should receive results similar to the following:

```
+--------------+
| FOUND_ROWS() |
+--------------+
|            5 |
+--------------+
1 row in set (0.00 sec)
```

How It Works

The first SELECT statement that you created includes the USER() function, which returns the name of the current user and host:

```
SELECT USER();
```

The next SELECT statement that you created includes the DATABASE() function, which returns the name of the current database:

```
SELECT DATABASE();
```

The third SELECT statement that you created includes the CONNECTION_ID() function, which returns the current connection ID:

```
SELECT CONNECTION_ID();
```

Next you created a SELECT statement that includes the SQL_CALC_FOUND_ROWS option and a LIMIT clause:

```
SELECT SQL_CALC_FOUND_ROWS DVDName
FROM DVDs
WHERE StatID='s2'
ORDER BY DVDName
LIMIT 2;
```

The SQL_CALC_FOUND_ROWS option stores the number of actual rows that would have been returned had you not used the LIMIT clause. Because you used the SQL_CALC_FOUND_ROWS option, you can now use the FOUND_ROWS() function:

```
SELECT FOUND_ROWS();
```

The SELECT statement returns the value that was stored as a result of specifying the SQL_CALC_FOUND_ROWS option. In this case, five rows would have been returned had not the LIMIT clause limited the result set to two rows.

Summary

This chapter introduced you to a number of functions that you can use to retrieve, extract, calculate, and summarize data. The chapter also provided numerous examples that demonstrated how to use the functions. Although the majority of these examples were SELECT statements, you can use most functions in any statement or statement clause that supports expressions. In addition, you can often use expressions as arguments in the functions themselves. The way in which you can use a function depends on the function itself. For this reason, the chapter provided a description of a variety of functions supported by MySQL. Specifically, the chapter described how to use functions to perform any of the following tasks:

- ❑ Use comparison functions to compare values in an expression and use cast functions to convert data to a different type.
- ❑ Use control flow functions to return results based on specifications in the functions.
- ❑ Use data-specific functions to perform operations on string, numerical, and date/time data.
- ❑ Use aggregate functions to summarize data grouped together in a SELECT statement.
- ❑ Use system-related functions to encrypt and decrypt data, view system information, and retrieve information about query and insert operations.

Although you learned how to use numerous types of functions in this chapter, you should consider reviewing the MySQL product documentation for additional information on any functions that you include in your code. In addition, whenever you upgrade to a newer version of MySQL, you should verify that the functions still perform as you would expect them to perform. Subtle differences can exist between versions of MySQL. Despite the difference, the basic functionality supported by the functions described in this chapter is fairly consistent from one version to the next. Chapter 10 covers accessing data in multiple tables through the use of joins, subqueries, and unions.

Exercises

For these exercises, you create a series of SELECT statements that use several of the functions described in this chapter. The statements are based on the Produce table, which is shown in 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,
   DateOrdered DATE NOT NULL
);
```

You can assume that the following INSERT statement populated the Produce table:

```
INSERT INTO Produce
VALUES (101, 'Apples', 'Red Delicious', 2000, 1000, '2004-10-12'),
(102, 'Apples', 'Fuji', 1500, 1200, '2004-10-11'),
(103, 'Apples', 'Golden Delicious', 500, 1000, '2004-10-12'),
(104, 'Apples', 'Granny Smith', 300, 800, '2004-10-12'),
(105, 'Oranges', 'Valencia', 1200, 1600, '2004-10-11'),
(106, 'Oranges', 'Seville', 1300, 1000, '2004-10-12'),
(107, 'Grapes', 'Red seedless', 3500, 1500, '2004-10-13'),
(108, 'Grapes', 'Green seedless', 3500, 1500, '2004-10-12'),
(109, 'Carrots', NULL, 4500, 1500, '2004-10-15'),
(110, 'Broccoli', NULL, 800, 2500, '2004-10-15'),
(111, 'Cherries', 'Bing', 2500, 2500, '2004-10-11'),
(112, 'Cherries', 'Rainier', 1500, 1500, '2004-10-12'),
(113, 'Zucchini', NULL, 1000, 1300, '2004-10-09'),
(114, 'Mushrooms', 'Shitake', 800, 900, '2004-10-10'),
(115, 'Mushrooms', 'Porcini', 400, 600, '2004-10-11'),
(116, 'Mushrooms', 'Portobello', 900, 1100, '2004-10-13'),
(117, 'Cucumbers', NULL, 2500, 1200, '2004-10-14');
```

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 and InStock columns. The result set should also include a column named Signage whose values depend on the values in the ProdName column. For apples, the Signage column should display a value that reads "On Sale!" For oranges, the Signage column should display a value that reads "Just Arrived!" For all other produce, the Signage column should display a value that reads "Fresh Crop!" In addition, the result set should include only those rows whose InStock value is greater than or equal to 1000, and the result set should be sorted according to the values in the ProdName column.
2. Create a SELECT statement that retrieves data from the ProdName, Variety, and InStock columns. The values in the InStock column should be converted to a CHAR type, and the column should be named InStock_CHAR. In addition, the result set should include only those rows whose InStock value is greater than or equal to 1000, and the result set should be sorted according to the values in the ProdName column.
3. Create a SELECT statement that retrieves data from the ProdName, Variety, and InStock columns. The values in the ProdName and Variety column should be concatenated and displayed in a column named ProduceVariety. The values in the column should be in the format <*ProdName*> (<*Variety*>). In addition, the result set should include only those rows whose InStock value is greater than or equal to 1000 and those rows whose Variety value is not NULL. Also, the result set should be sorted according to the values in the ProdName column.
4. Create a SELECT statement that is identical to the one in Exercise 3, only return the ProdName and Variety values in all uppercase.
5. Create a SELECT statement that retrieves data from the Variety, OnOrder, and DateOrdered columns. The result set should also include a column named DeliveryDate, which should include values that add four days to the DateOrdered values. In addition, the result set should include only rows for apples, and the rows should be sorted according to the variety of the apples.