Using Functions in Your SQL Statements

- We will explain the purpose of each of these functions, describe the results you can expect when a statement includes a function, and provide examples that demonstrate how to use each function.

- This lecture doesn't describe every function included with MySQL, but it covers many of them that you're most likely to use when creating SQL statements.
  - Comparison, control flow, and cast functions 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 3 types of functions that allow you to compare and convert data:
  - comparison functions
  - control flow functions
  - cast functions

1. Comparison Functions

- Comparison functions are similar to the comparison operators we discussed before.
- 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.

GREATEST() Function

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);
```

LEAST() Functions

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() Function

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.

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.

INTERVAL()

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.

INTERVAL()

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-Class Exercise

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

```
use DVDRentals
```

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';
```

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';
```

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;
```

2. Control Flow Functions

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');
```

IFNULL() Function

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.

NULLIF() Function

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;
```

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;
```

In-Class Exercise

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

```
use DVDRentals
```

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;
```

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;
```

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;
```

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);
```

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.)

CAST() and CONVERT functions

- The CAST() and CONVERT() 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.

```
SELECT (CAST(field1 AS SIGNED)) + 10 FROM myTable;

SELECT (CONVERT(field1,SIGNED)) + 10 FROM myTable;
```

Managing Different Types of Data

- MySQL includes functions to manage string, numeric, and date/time data.
- Unlike the functions that we have 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.

String Functions - ASCII

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.

String Functions - ORD

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.

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).

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`.

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>)
```

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);
```

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.

Date/Time Functions

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

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>'
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>'

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 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.

Example

- 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.
- Suppose 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.

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();
```

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.

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.

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.

Summary Functions

- AVG() Function
- SUM() Function
- MIN() Function
- MAX() Function
- COUNT() Function