CHAPTER 4

Filtering

Sometimes you will want to work with every row in a table, such as:

- Purging all data from a table used to stage new data warehouse feeds
- Modifying all rows in a table after a new column has been added
- Retrieving all rows from a message queue table

In cases like these, your SQL statements won't need to have a `where` clause, since you don't need to exclude any rows from consideration. Most of the time, however, you will want to narrow your focus to a subset of a table's rows. Therefore, all the SQL data statements (except the `insert` statement) include an optional `where` clause to house *filter conditions* used to restrict the number of rows acted on by the SQL statement. Additionally, the `select` statement includes a `having` clause in which filter conditions pertaining to grouped data may be included. This chapter explores the various types of filter conditions that you can employ in the `where` clauses of `select`, `update`, and `delete` statements; we explore the use of filter conditions in the `having` clause of a `select` statement in Chapter 8.

Condition Evaluation

A `where` clause may contain one or more *conditions*, separated by the operators `and` and `or`. If multiple conditions are separated only by the `and` operator, then all the conditions must evaluate to `true` for the row to be included in the result set. Consider the following `where` clause:

```
WHERE title = 'Teller' AND start_date < '2007-01-01'
```

Given these two conditions, only tellers who began working for the bank prior to 2007 will be included (or, to look at it another way, any employee who is either not a teller or began working for the bank in 2007 or later will be removed from consideration). Although this example uses only two conditions, no matter how many conditions are in your `where` clause, if they are separated by the `and` operator they must *all* evaluate to `true` for the row to be included in the result set.

If all conditions in the `where` clause are separated by the `or` operator, however, only *one* of the conditions must evaluate to `true` for the row to be included in the result set. Consider the following two conditions:

```
WHERE title = 'Teller' OR start_date < '2007-01-01'
```

There are now various ways for a given `employee` row to be included in the result set:

- The employee is a teller and was employed prior to 2007.
- The employee is a teller and was employed after January 1, 2007.
- The employee is something other than a teller but was employed prior to 2007.

Table 4-1 shows the possible outcomes for a `where` clause containing two conditions separated by the `or` operator.

Table 4-1. Two-condition evaluation using or

Intermediate result	Final result
`WHERE true OR true`	`True`
`WHERE true OR false`	`True`
`WHERE false OR true`	`True`
`WHERE false OR false`	`False`

In the case of the preceding example, the only way for a row to be excluded from the result set is if the employee is not a teller and was employed on or after January 1, 2007.

Using Parentheses

If your `where` clause includes three or more conditions using both the `and` and `or` operators, you should use parentheses to make your intent clear, both to the database server and to anyone else reading your code. Here's a `where` clause that extends the previous example by checking to make sure that the employee is still employed by the bank:

```
WHERE end_date IS NULL
  AND (title = 'Teller' OR start_date < '2007-01-01')
```

There are now three conditions; for a row to make it to the final result set, the first condition must evaluate to `true`, and either the second *or* third condition (or both) must evaluate to `true`. Table 4-2 shows the possible outcomes for this `where` clause.

Table 4-2. Three-condition evaluation using and, or

Intermediate result	Final result
`WHERE true AND (true OR true)`	`True`
`WHERE true AND (true OR false)`	`True`
`WHERE true AND (false OR true)`	`True`
`WHERE true AND (false OR false)`	`False`

Intermediate result	Final result
`WHERE false AND (true OR true)`	False
`WHERE false AND (true OR false)`	False
`WHERE false AND (false OR true)`	False
`WHERE false AND (false OR false)`	False

As you can see, the more conditions you have in your `where` clause, the more combinations there are for the server to evaluate. In this case, only three of the eight combinations yield a final result of `true`.

Using the not Operator

Hopefully, the previous three-condition example is fairly easy to understand. Consider the following modification, however:

```
WHERE end_date IS NULL
  AND NOT (title = 'Teller' OR start_date < '2007-01-01')
```

Did you spot the change from the previous example? I added the `not` operator after the `and` operator on the second line. Now, instead of looking for nonterminated employees who either are tellers or began working for the bank prior to 2007, I am looking for nonterminated employees who both are nontellers and began working for the bank in 2007 or later. Table 4-3 shows the possible outcomes for this example.

Table 4-3. Three-condition evaluation using and, or, and not

Intermediate result	Final result
`WHERE true AND NOT (true OR true)`	False
`WHERE true AND NOT (true OR false)`	False
`WHERE true AND NOT (false OR true)`	False
`WHERE true AND NOT (false OR false)`	True
`WHERE false AND NOT (true OR true)`	False
`WHERE false AND NOT (true OR false)`	False
`WHERE false AND NOT (false OR true)`	False
`WHERE false AND NOT (false OR false)`	False

While it is easy for the database server to handle, it is typically difficult for a person to evaluate a `where` clause that includes the `not` operator, which is why you won't encounter it very often. In this case, you can rewrite the `where` clause to avoid using the `not` operator:

```
WHERE end_date IS NULL
  AND title != 'Teller' AND start_date >= '2007-01-01'
```

While I'm sure that the server doesn't have a preference, you probably have an easier time understanding this version of the `where` clause.

Building a Condition

Now that you have seen how the server evaluates multiple conditions, let's take a step back and look at what comprises a single condition. A condition is made up of one or more *expressions* coupled with one or more *operators*. An expression can be any of the following:

- A number
- A column in a table or view
- A string literal, such as `'Teller'`
- A built-in function, such as `concat('Learning', ' ', 'SQL')`
- A subquery
- A list of expressions, such as `('Teller', 'Head Teller', 'Operations Manager')`

The operators used within conditions include:

- Comparison operators, such as `=`, `!=`, `<`, `>`, `<>`, `LIKE`, `IN`, and `BETWEEN`
- Arithmetic operators, such as `+`, `-`, `*`, and `/`

The following section demonstrates how you can combine these expressions and operators to manufacture the various types of conditions.

Condition Types

There are many different ways to filter out unwanted data. You can look for specific values, sets of values, or ranges of values to include or exclude, or you can use various pattern-searching techniques to look for partial matches when dealing with string data. The next four subsections explore each of these condition types in detail.

Equality Conditions

A large percentage of the filter conditions that you write or come across will be of the form `'`*`column = expression`*`'` as in:

```
title = 'Teller'
fed_id = '111-11-1111'
amount = 375.25
dept_id = (SELECT dept_id FROM department WHERE name = 'Loans')
```

Conditions such as these are called *equality conditions* because they equate one expression to another. The first three examples equate a column to a literal (two strings and a number), and the fourth example equates a column to the value returned from a subquery. The following query uses two equality conditions; one in the `on` clause (a join condition), and the other in the `where` clause (a filter condition):

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
    ->   ON p.product_type_cd = pt.product_type_cd
    -> WHERE pt.name = 'Customer Accounts';
+-------------------+------------------------+
| product_type      | product                |
+-------------------+------------------------+
| Customer Accounts | certificate of deposit |
| Customer Accounts | checking account       |
| Customer Accounts | money market account   |
| Customer Accounts | savings account        |
+-------------------+------------------------+
4 rows in set (0.08 sec)
```

This query shows all products that are *customer account* types.

Inequality conditions

Another fairly common type of condition is the *inequality condition*, which asserts that two expressions are *not* equal. Here's the previous query with the filter condition in the `where` clause changed to an inequality condition:

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
    ->   ON p.product_type_cd = pt.product_type_cd
    -> WHERE pt.name <> 'Customer Accounts';
+-------------------------------+-------------------------+
| product_type                  | product                 |
+-------------------------------+-------------------------+
| Individual and Business Loans | auto loan               |
| Individual and Business Loans | business line of credit |
| Individual and Business Loans | home mortgage           |
| Individual and Business Loans | small business loan     |
+-------------------------------+-------------------------+
4 rows in set (0.00 sec)
```

This query shows all products that are *not* customer account types. When building inequality conditions, you may choose to use either the `!=` or `<>` operator.

Data modification using equality conditions

Equality/inequality conditions are commonly used when modifying data. For example, let's say that the bank has a policy of removing old account rows once per year. Your task is to remove rows from the `account` table that were closed in 2002. Here's one way to tackle it:

```
DELETE FROM account
WHERE status = 'CLOSED' AND YEAR(close_date) = 2002;
```

This statement includes two equality conditions: one to find only closed accounts, and another to check for those accounts closed in 2002.

When crafting examples of `delete` and `update` statements, I try to write each statement such that no rows are modified. That way, when you execute the statements, your data will remain unchanged, and your output from `select` statements will always match that shown in this book.

Since MySQL sessions are in auto-commit mode by default (see Chapter 12), you would not be able to roll back (undo) any changes made to the example data if one of my statements modified the data. You may, of course, do whatever you want with the example data, including wiping it clean and rerunning the scripts I have provided, but I try to leave it intact.

Range Conditions

Along with checking that an expression is equal to (or not equal to) another expression, you can build conditions that check whether an expression falls within a certain range. This type of condition is common when working with numeric or temporal data. Consider the following query:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date < '2007-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     11 | Thomas  | Ziegler   | 2004-10-23 |
|     13 | John    | Blake     | 2004-05-11 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
13 rows in set (0.15 sec)
```

This query finds all employees hired prior to 2007. Along with specifying an upper limit for the start date, you may also want to specify a lower range for the start date:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date < '2007-01-01'
    ->   AND start_date >= '2005-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
11 rows in set (0.00 sec)
```

This version of the query retrieves all employees hired in 2005 or 2006.

The between operator

When you have *both* an upper and lower limit for your range, you may choose to use a single condition that utilizes the between operator rather than using two separate conditions, as in:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date BETWEEN '2005-01-01' AND '2007-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
11 rows in set (0.03 sec)
```

When using the between operator, there are a couple of things to keep in mind. You should always specify the lower limit of the range first (after between) and the upper limit of the range second (after and). Here's what happens if you mistakenly specify the upper limit first:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date BETWEEN '2007-01-01' AND '2005-01-01';
Empty set (0.00 sec)
```

As you can see, no data is returned. This is because the server is, in effect, generating two conditions from your single condition using the <= and >= operators, as in:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date >= '2007-01-01'
    ->   AND start_date <= '2005-01-01';
Empty set (0.00 sec)
```

Since it is impossible to have a date that is *both* greater than January 1, 2007 and less than January 1, 2005, the query returns an empty set. This brings me to the second pitfall when using `between`, which is to remember that your upper and lower limits are *inclusive*, meaning that the values you provide are included in the range limits. In this case, I want to specify `2005-01-01` as the lower end of the range and `2006-12-31` as the upper end, rather than `2007-01-01`. Even though there probably weren't any employees who started working for the bank on New Year's Day 2007, it is best to specify exactly what you want.

Along with dates, you can also build conditions to specify ranges of numbers. Numeric ranges are fairly easy to grasp, as demonstrated by the following:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE avail_balance BETWEEN 3000 AND 5000;
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          3 | CD         |       1 |       3000.00 |
|         17 | CD         |       7 |       5000.00 |
|         18 | CHK        |       8 |       3487.19 |
+------------+------------+---------+---------------+
3 rows in set (0.10 sec)
```

All accounts with between $3,000 and $5,000 of an available balance are returned. Again, make sure that you specify the lower amount first.

String ranges

While ranges of dates and numbers are easy to understand, you can also build conditions that search for ranges of strings, which are a bit harder to visualize. Say, for example, you are searching for customers having a Social Security number that falls within a certain range. The format for a Social Security number is "XXX-XX-XXXX," where X is a number from 0 to 9, and you want to find every customer whose Social Security number lies between "500-00-0000" and "999-99-9999." Here's what the statement would look like:

```
mysql> SELECT cust_id, fed_id
    -> FROM customer
    -> WHERE cust_type_cd = 'I'
    ->   AND fed_id BETWEEN '500-00-0000' AND '999-99-9999';
+---------+-------------+
| cust_id | fed_id      |
+---------+-------------+
|       5 | 555-55-5555 |
|       6 | 666-66-6666 |
|       7 | 777-77-7777 |
|       8 | 888-88-8888 |
|       9 | 999-99-9999 |
+---------+-------------+
5 rows in set (0.01 sec)
```

To work with string ranges, you need to know the order of the characters within your character set (the order in which the characters within a character set are sorted is called a *collation*).

Membership Conditions

In some cases, you will not be restricting an expression to a single value or range of values, but rather to a finite set of values. For example, you might want to locate all accounts whose product code is either `'CHK'`, `'SAV'`, `'CD'`, or `'MM'`:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd = 'CHK' OR product_cd = 'SAV'
    ->   OR product_cd = 'CD' OR product_cd = 'MM';
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          1 | CHK        |       1 |       1057.75 |
|          2 | SAV        |       1 |        500.00 |
|          3 | CD         |       1 |       3000.00 |
|          4 | CHK        |       2 |       2258.02 |
|          5 | SAV        |       2 |        200.00 |
|          7 | CHK        |       3 |       1057.75 |
|          8 | MM         |       3 |       2212.50 |
|         10 | CHK        |       4 |        534.12 |
|         11 | SAV        |       4 |        767.77 |
|         12 | MM         |       4 |       5487.09 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         15 | CD         |       6 |      10000.00 |
|         17 | CD         |       7 |       5000.00 |
|         18 | CHK        |       8 |       3487.19 |
|         19 | SAV        |       8 |        387.99 |
|         21 | CHK        |       9 |        125.67 |
|         22 | MM         |       9 |       9345.55 |
|         23 | CD         |       9 |       1500.00 |
|         24 | CHK        |      10 |      23575.12 |
|         28 | CHK        |      12 |      38552.05 |
+------------+------------+---------+---------------+
21 rows in set (0.28 sec)
```

While this `where` clause (four conditions `or`'d together) wasn't too tedious to generate, imagine if the set of expressions contained 10 or 20 members. For these situations, you can use the `in` operator instead:

```
SELECT account_id, product_cd, cust_id, avail_balance
FROM account
WHERE product_cd IN ('CHK','SAV','CD','MM');
```

With the `in` operator, you can write a single condition no matter how many expressions are in the set.

Using subqueries

Along with writing your own set of expressions, such as `('CHK','SAV','CD','MM')`, you can use a subquery to generate a set for you on the fly. For example, all four product types used in the previous query have a `product_type_cd` of `'ACCOUNT'`, so why not use a subquery against the `product` table to retrieve the four product codes instead of explicitly naming them:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd IN (SELECT product_cd FROM product
    ->   WHERE product_type_cd = 'ACCOUNT');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          3 | CD         |       1 |       3000.00 |
|         15 | CD         |       6 |      10000.00 |
|         17 | CD         |       7 |       5000.00 |
|         23 | CD         |       9 |       1500.00 |
|          1 | CHK        |       1 |       1057.75 |
|          4 | CHK        |       2 |       2258.02 |
|          7 | CHK        |       3 |       1057.75 |
|         10 | CHK        |       4 |        534.12 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         18 | CHK        |       8 |       3487.19 |
|         21 | CHK        |       9 |        125.67 |
|         24 | CHK        |      10 |      23575.12 |
|         28 | CHK        |      12 |      38552.05 |
|          8 | MM         |       3 |       2212.50 |
|         12 | MM         |       4 |       5487.09 |
|         22 | MM         |       9 |       9345.55 |
|          2 | SAV        |       1 |        500.00 |
|          5 | SAV        |       2 |        200.00 |
|         11 | SAV        |       4 |        767.77 |
|         19 | SAV        |       8 |        387.99 |
+------------+------------+---------+---------------+
21 rows in set (0.11 sec)
```

The subquery returns a set of four values, and the main query checks to see whether the value of the `product_cd` column can be found in the set that the subquery returned.

Using not in

Sometimes you want to see whether a particular expression exists within a set of expressions, and sometimes you want to see whether the expression does *not* exist. For these situations, you can use the `not in` operator:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd NOT IN ('CHK','SAV','CD','MM');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         25 | BUS        |      10 |          0.00 |
|         27 | BUS        |      11 |       9345.55 |
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
3 rows in set (0.09 sec)
```

This query finds all accounts that are *not* checking, savings, certificate of deposit, or money market accounts.

Matching Conditions

So far, you have been introduced to conditions that identify an exact string, a range of strings, or a set of strings; the final condition type deals with partial string matches. You may, for example, want to find all employees whose last name begins with *T*. You could use a built-in function to strip off the first letter of the `lname` column, as in:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE LEFT(lname, 1) = 'T';
+--------+--------+--------+
| emp_id | fname  | lname  |
+--------+--------+--------+
|      3 | Robert | Tyler  |
|      7 | Chris  | Tucker |
|     18 | Rick   | Tulman |
+--------+--------+--------+
3 rows in set (0.01 sec)
```

While the built-in function `left()` does the job, it doesn't give you much flexibility. Instead, you can use wildcard characters to build search expressions, as demonstrated in the next section.

Using wildcards

When searching for partial string matches, you might be interested in:

- Strings beginning/ending with a certain character
- Strings beginning/ending with a substring
- Strings containing a certain character anywhere within the string
- Strings containing a substring anywhere within the string
- Strings with a specific format, regardless of individual characters

You can build search expressions to identify these and many other partial string matches by using the wildcard characters shown in Table 4-4.

Table 4-4. Wildcard characters

Wildcard character	Matches
_	Exactly one character
%	Any number of characters (including 0)

The underscore character takes the place of a single character, while the percent sign can take the place of a variable number of characters. When building conditions that utilize search expressions, you use the `like` operator, as in:

```
mysql> SELECT lname
    -> FROM employee
    -> WHERE lname LIKE '_a%e%';
+-----------+
| lname     |
+-----------+
| Barker    |
| Hawthorne |
| Parker    |
| Jameson   |
+-----------+
4 rows in set (0.00 sec)
```

The search expression in the previous example specifies strings containing an *a* in the second position and followed by an *e* at any other position in the string (including the last position). Table 4-5 shows some more search expressions and their interpretations.

Table 4-5. Sample search expressions

Search expression	Interpretation
F%	Strings beginning with *F*
%t	Strings ending with *t*
%bas%	Strings containing the substring `'bas'`
_ _t_	Four-character strings with a *t* in the third position
_ _ _-_ _-_ _ _ _	11-character strings with dashes in the fourth and seventh positions

You could use the last example in Table 4-5 to find customers whose federal ID matches the format used for Social Security numbers, as in:

```
mysql> SELECT cust_id, fed_id
    -> FROM customer
    -> WHERE fed_id LIKE '___-__-____';
+---------+-------------+
| cust_id | fed_id      |
+---------+-------------+
|       1 | 111-11-1111 |
|       2 | 222-22-2222 |
|       3 | 333-33-3333 |
|       4 | 444-44-4444 |
|       5 | 555-55-5555 |
|       6 | 666-66-6666 |
|       7 | 777-77-7777 |
|       8 | 888-88-8888 |
|       9 | 999-99-9999 |
+---------+-------------+
9 rows in set (0.02 sec)
```

The wildcard characters work fine for building simple search expressions; if your needs are a bit more sophisticated, however, you can use multiple search expressions, as demonstrated by the following:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname LIKE 'F%' OR lname LIKE 'G%';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

This query finds all employees whose last name begins with *F* or *G*.

Using regular expressions

If you find that the wildcard characters don't provide enough flexibility, you can use regular expressions to build search expressions. A regular expression is, in essence, a search expression on steroids. If you are new to SQL but have coded using programming languages such as Perl, then you might already be intimately familiar with regular expressions. If you have never used regular expressions, then you may want to consult Jeffrey E.F. Friedl's *Mastering Regular Expressions (http://oreilly.com/catalog/9780596528126/)* (O'Reilly), since it is far too large a topic to try to cover in this book.

Here's what the previous query (find all employees whose last name starts with *F* or *G*) would look like using the MySQL implementation of regular expressions:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname REGEXP '^[FG]';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

The `regexp` operator takes a regular expression (`'^[FG]'` in this example) and applies it to the expression on the lefthand side of the condition (the column `lname`). The query now contains a single condition using a regular expression rather than two conditions using wildcard characters.

Oracle Database and Microsoft SQL Server also support regular expressions. With Oracle Database, you would use the `regexp_like` function instead of the `regexp` operator shown in the previous example, whereas SQL Server allows regular expressions to be used with the `like` operator.

Null: That Four-Letter Word

I put it off as long as I could, but it's time to broach a topic that tends to be met with fear, uncertainty, and dread: the `null` value. `Null` is the absence of a value; before an employee is terminated, for example, her `end_date` column in the `employee` table should be `null`. There is no value that can be assigned to the `end_date` column that would make sense in this situation. `Null` is a bit slippery, however, as there are various flavors of `null`:

Not applicable
Such as the employee ID column for a transaction that took place at an ATM machine

Value not yet known
Such as when the federal ID is not known at the time a customer row is created

Value undefined
Such as when an account is created for a product that has not yet been added to the database

Some theorists argue that there should be a different expression to cover each of these (and more) situations, but most practitioners would agree that having multiple `null` values would be far too confusing.

When working with `null`, you should remember:

- An expression can *be* null, but it can never *equal* null.
- Two nulls are never equal to each other.

To test whether an expression is `null`, you need to use the `is null` operator, as demonstrated by the following:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id IS NULL;
+--------+---------+-------+-----------------+
| emp_id | fname   | lname | superior_emp_id |
+--------+---------+-------+-----------------+
|      1 | Michael | Smith |            NULL |
+--------+---------+-------+-----------------+
1 row in set (0.00 sec)
```

This query returns all employees who do not have a boss (wouldn't that be nice?). Here's the same query using `= null` instead of `is null`:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id = NULL;
Empty set (0.01 sec)
```

As you can see, the query parses and executes but does not return any rows. This is a common mistake made by inexperienced SQL programmers, and the database server will not alert you to your error, so be careful when constructing conditions that test for `null`.

If you want to see whether a value has been assigned to a column, you can use the `is not null` operator, as in:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id IS NOT NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|      7 | Chris    | Tucker    |               6 |
|      8 | Sarah    | Parker    |               6 |
|      9 | Jane     | Grossman  |               6 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
17 rows in set (0.00 sec)
```

This version of the query returns the other 17 employees who, unlike Michael Smith, have a boss.

Before putting `null` aside for a while, it would be helpful to investigate one more potential pitfall. Suppose that you have been asked to identify all employees who are *not* managed by Helen Fleming (whose employee ID is 6). Your first instinct might be to do the following:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id != 6;
```

```
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
14 rows in set (0.00 sec)
```

While it is true that these 14 employees do not work for Helen Fleming, if you look carefully at the data, you will see that there is one more employee who doesn't work for Helen who is not listed here. That employee is Michael Smith, and his `superior_emp_id` column is `null` (because he's the big cheese). To answer the question correctly, therefore, you need to account for the possibility that some rows might contain a `null` in the `superior_emp_id` column:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id != 6 OR superior_emp_id IS NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      1 | Michael  | Smith     |            NULL |
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
15 rows in set (0.00 sec)
```

The result set now includes all 15 employees who don't work for Helen. When working with a database that you are not familiar with, it is a good idea to find out which columns