CHAPTER 3
Query Primer

So far, you have seen a few examples of database queries (a.k.a. `select` statements) sprinkled throughout the first two chapters. Now it's time to take a closer look at the different parts of the `select` statement and how they interact.

Query Mechanics

Before dissecting the `select` statement, it might be interesting to look at how queries are executed by the MySQL server (or, for that matter, any database server). If you are using the `mysql` command-line tool (which I assume you are), then you have already logged in to the MySQL server by providing your username and password (and possibly a hostname if the MySQL server is running on a different computer). Once the server has verified that your username and password are correct, a *database connection* is generated for you to use. This connection is held by the application that requested it (which, in this case, is the `mysql` tool) until the application releases the connection (i.e., as a result of your typing `quit`) or the server closes the connection (i.e., when the server is shut down). Each connection to the MySQL server is assigned an identifier, which is shown to you when you first log in:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 6.0.3-alpha-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

In this case, my connection ID is `11`. This information might be useful to your database administrator if something goes awry, such as a malformed query that runs for hours, so you might want to jot it down.

Once the server has verified your username and password and issued you a connection, you are ready to execute queries (along with other SQL statements). Each time a query is sent to the server, the server checks the following things prior to statement execution:

- Do you have permission to execute the statement?
- Do you have permission to access the desired data?
- Is your statement syntax correct?

If your statement passes these three tests, then your query is handed to the *query optimizer*, whose job it is to determine the most efficient way to execute your query. The optimizer will look at such things as the order in which to join the tables named in your `from` clause and what indexes are available, and then picks an *execution plan*, which the server uses to execute your query.

> Understanding and influencing how your database server chooses execution plans is a fascinating topic that many of you will wish to explore. For those readers using MySQL, you might consider reading Baron Schwartz et al.'s *High Performance MySQL (http://oreilly.com/catalog/9780596101718/)* (O'Reilly). Among other things, you will learn how to generate indexes, analyze execution plans, influence the optimizer via query hints, and tune your server's startup parameters. If you are using Oracle Database or SQL Server, dozens of tuning books are available.

Once the server has finished executing your query, the *result set* is returned to the calling application (which is, once again, the `mysql` tool). As I mentioned in Chapter 1, a result set is just another table containing rows and columns. If your query fails to yield any results, the `mysql` tool will show you the message found at the end of the following example:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname = 'Bkadfl';
Empty set(0.00 sec)
```

If the query returns one or more rows, the `mysql` tool will format the results by adding column headers and by constructing boxes around the columns using the -, |, and + symbols, as shown in the next example:

```
mysql> SELECT fname, lname
    -> FROM employee;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| Michael  | Smith     |
| Susan    | Barker    |
| Robert   | Tyler     |
| Susan    | Hawthorne |
| John     | Gooding   |
| Helen    | Fleming   |
| Chris    | Tucker    |
| Sarah    | Parker    |
| Jane     | Grossman  |
| Paula    | Roberts   |
| Thomas   | Ziegler   |
| Samantha | Jameson   |
| John     | Blake     |
| Cindy    | Mason     |
| Frank    | Portman   |
| Theresa  | Markham   |
| Beth     | Fowler    |
| Rick     | Tulman    |
+----------+-----------+
18 rows in set (0.00 sec)
```

This query returns the first and last names of all the employees in the `employee` table. After the last row of data is displayed, the `mysql` tool displays a message telling you how many rows were returned, which, in this case, is 18.

Query Clauses

Several components or *clauses* make up the `select` statement. While only one of them is mandatory when using MySQL (the `select` clause), you will usually include at least two or three of the six available clauses. Table 3-1 shows the different clauses and their purposes.

Table 3-1. Query clauses

Clause name	Purpose
`Select`	Determines which columns to include in the query's result set
`From`	Identifies the tables from which to draw data and how the tables should be joined
`Where`	Filters out unwanted data
`Group by`	Used to group rows together by common column values
`Having`	Filters out unwanted groups
`Order by`	Sorts the rows of the final result set by one or more columns

All of the clauses shown in Table 3-1 are included in the ANSI specification; additionally, several other clauses are unique to MySQL that we explore in Appendix B. The following sections delve into the uses of the six major query clauses.

The select Clause

Even though the `select` clause is the first clause of a `select` statement, it is one of the last clauses that the database server evaluates. The reason for this is that before you can determine what to include in the final result set, you need to know all of the possible columns that *could* be included in the final result set. In order to fully understand the role of the `select` clause, therefore, you will need to understand a bit about the `from` clause. Here's a query to get started:

```
mysql> SELECT *
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
+---------+----------------+
3 rows in set (0.04 sec)
```

In this query, the `from` clause lists a single table (`department`), and the `select` clause indicates that *all* columns (designated by *) in the `department` table should be included in the result set. This query could be described in English as follows:

Show me all the columns and all the rows in the `department` *table.*

In addition to specifying all the columns via the asterisk character, you can explicitly name the columns you are interested in, such as:

```
mysql> SELECT dept_id, name
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
+---------+----------------+
3 rows in set (0.01 sec)
```

The results are identical to the first query, since all the columns in the `department` table (`dept_id` and `name`) are named in the `select` clause. You can choose to include only a subset of the columns in the `department` table as well:

```
mysql> SELECT name
    -> FROM department;
+----------------+
| name           |
+----------------+
| Operations     |
| Loans          |
| Administration |
+----------------+
3 rows in set (0.00 sec)
```

The job of the `select` clause, therefore, is the following:

The `select` *clause determines which of all possible columns should be included in the query's result set.*

If you were limited to including only columns from the table or tables named in the `from` clause, things would be rather dull. However, you can spice things up by including in your `select` clause such things as:

- Literals, such as numbers or strings
- Expressions, such as `transaction.amount * -1`
- Built-in function calls, such as `ROUND(transaction.amount, 2)`
- User-defined function calls

The next query demonstrates the use of a table column, a literal, an expression, and a built-in function call in a single query against the `employee` table:

```
mysql> SELECT emp_id,
    ->  'ACTIVE',
    ->  emp_id * 3.14159,
    ->  UPPER(lname)
    -> FROM employee;
+--------+--------+------------------+--------------+
| emp_id | ACTIVE | emp_id * 3.14159 | UPPER(lname) |
+--------+--------+------------------+--------------+
|      1 | ACTIVE |          3.14159 | SMITH        |
|      2 | ACTIVE |          6.28318 | BARKER       |
|      3 | ACTIVE |          9.42477 | TYLER        |
|      4 | ACTIVE |         12.56636 | HAWTHORNE    |
|      5 | ACTIVE |         15.70795 | GOODING      |
|      6 | ACTIVE |         18.84954 | FLEMING      |
|      7 | ACTIVE |         21.99113 | TUCKER       |
|      8 | ACTIVE |         25.13272 | PARKER       |
|      9 | ACTIVE |         28.27431 | GROSSMAN     |
|     10 | ACTIVE |         31.41590 | ROBERTS      |
|     11 | ACTIVE |         34.55749 | ZIEGLER      |
|     12 | ACTIVE |         37.69908 | JAMESON      |
|     13 | ACTIVE |         40.84067 | BLAKE        |
|     14 | ACTIVE |         43.98226 | MASON        |
|     15 | ACTIVE |         47.12385 | PORTMAN      |
|     16 | ACTIVE |         50.26544 | MARKHAM      |
|     17 | ACTIVE |         53.40703 | FOWLER       |
|     18 | ACTIVE |         56.54862 | TULMAN       |
+--------+--------+------------------+--------------+
18 rows in set (0.05 sec)
```

We cover expressions and built-in functions in detail later, but I wanted to give you a feel for what kinds of things can be included in the `select` clause. If you only need to execute a built-in function or evaluate a simple expression, you can skip the `from` clause entirely. Here's an example:

```
mysql> SELECT VERSION(),
    ->   USER(),
    ->   DATABASE();
+-----------------------+-------------------+------------+
| version()             | user()            | database() |
+-----------------------+-------------------+------------+
| 6.0.3-alpha-community | lrngsql@localhost | bank       |
+-----------------------+-------------------+------------+
1 row in set (0.05 sec)
```

Since this query simply calls three built-in functions and doesn't retrieve data from any tables, there is no need for a `from` clause.

Column Aliases

Although the `mysql` tool will generate labels for the columns returned by your queries, you may want to assign your own labels. While you might want to assign a new label to a column from a table (if it is poorly or ambiguously named), you will almost certainly want to assign your own labels to those columns in your result set that are generated by expressions or built-in function calls. You can do so by adding a *column alias* after each element of your `select` clause. Here's the previous query against the `employee` table with column aliases applied to three of the columns:

```
mysql> SELECT emp_id,
    ->   'ACTIVE' status,
    ->   emp_id * 3.14159 empid_x_pi,
    ->   UPPER(lname) last_name_upper
    -> FROM employee;
+--------+--------+------------+-----------------+
| emp_id | status | empid_x_pi | last_name_upper |
+--------+--------+------------+-----------------+
|      1 | ACTIVE |    3.14159 | SMITH           |
|      2 | ACTIVE |    6.28318 | BARKER          |
|      3 | ACTIVE |    9.42477 | TYLER           |
|      4 | ACTIVE |   12.56636 | HAWTHORNE       |
|      5 | ACTIVE |   15.70795 | GOODING         |
|      6 | ACTIVE |   18.84954 | FLEMING         |
|      7 | ACTIVE |   21.99113 | TUCKER          |
|      8 | ACTIVE |   25.13272 | PARKER          |
|      9 | ACTIVE |   28.27431 | GROSSMAN        |
|     10 | ACTIVE |   31.41590 | ROBERTS         |
|     11 | ACTIVE |   34.55749 | ZIEGLER         |
|     12 | ACTIVE |   37.69908 | JAMESON         |
|     13 | ACTIVE |   40.84067 | BLAKE           |
|     14 | ACTIVE |   43.98226 | MASON           |
|     15 | ACTIVE |   47.12385 | PORTMAN         |
|     16 | ACTIVE |   50.26544 | MARKHAM         |
|     17 | ACTIVE |   53.40703 | FOWLER          |
|     18 | ACTIVE |   56.54862 | TULMAN          |
+--------+--------+------------+-----------------+
18 rows in set (0.00 sec)
```

If you look at the column headers, you can see that the second, third, and fourth columns now have reasonable names instead of simply being labeled with the function or expression that generated the column. If you look at the `select` clause, you can see how the column aliases `status`, `empid_x_pi`, and `last_name_upper` are added after the second, third, and fourth columns. I think you will agree that the output is easier to understand with column aliases in place, and it would be easier to work with programmatically if you were issuing the query from within Java or C# rather than interactively via the mysql tool. In order to make your column aliases stand out even more, you also have the option of using the as keyword before the alias name, as in:

```
mysql> SELECT emp_id,
    ->   'ACTIVE' AS status,
    ->   emp_id * 3.14159 AS empid_x_pi,
    ->   UPPER(lname) AS last_name_upper
    -> FROM employee;
```

Many people feel that including the optional as keyword improves readability, although I have chosen not to use it for the examples in this book.

Removing Duplicates

In some cases, a query might return duplicate rows of data. For example, if you were to retrieve the IDs of all customers that have accounts, you would see the following:

```
mysql> SELECT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       1 |
|       1 |
|       2 |
|       2 |
|       3 |
|       3 |
|       4 |
|       4 |
|       4 |
|       5 |
|       6 |
|       6 |
|       7 |
|       8 |
|       8 |
|       9 |
|       9 |
|       9 |
|      10 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
24 rows in set (0.00 sec)
```

Since some customers have more than one account, you will see the same customer ID once for each account owned by that customer. What you probably want in this case is the *distinct* set of customers that have accounts, instead of seeing the customer ID for each row in the `account` table. You can achieve this by adding the keyword `distinct` directly after the `select` keyword, as demonstrated by the following:

```
mysql> SELECT DISTINCT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
13 rows in set (0.01 sec)
```

The result set now contains 13 rows, one for each distinct customer, rather than 24 rows, one for each account.

If you do not want the server to remove duplicate data, or you are sure there will be no duplicates in your result set, you can specify the `ALL` keyword instead of specifying `DISTINCT`. However, the `ALL` keyword is the default and never needs to be explicitly named, so most programmers do not include `ALL` in their queries.

Keep in mind that generating a distinct set of results requires the data to be sorted, which can be time-consuming for large result sets. Don't fall into the trap of using `DISTINCT` just to be sure there are no duplicates; instead, take the time to understand the data you are working with so that you will know whether duplicates are possible.

The from Clause

Thus far, you have seen queries whose `from` clauses contain a single table. Although most SQL books will define the `from` clause as simply a list of one or more tables, I would like to broaden the definition as follows:

The `from` *clause defines the tables used by a query, along with the means of linking the tables together.*

This definition is composed of two separate but related concepts, which we explore in the following sections.

Tables

When confronted with the term *table*, most people think of a set of related rows stored in a database. While this does describe one type of table, I would like to use the word in a more general way by removing any notion of how the data might be stored and concentrating on just the set of related rows. Three different types of tables meet this relaxed definition:

- Permanent tables (i.e., created using the `create table` statement)
- Temporary tables (i.e., rows returned by a subquery)
- Virtual tables (i.e., created using the `create view` statement)

Each of these table types may be included in a query's `from` clause. By now, you should be comfortable with including a permanent table in a `from` clause, so I briefly describe the other types of tables that can be referenced in a `from` clause.

Subquery-generated tables

A subquery is a query contained within another query. Subqueries are surrounded by parentheses and can be found in various parts of a `select` statement; within the `from` clause, however, a subquery serves the role of generating a temporary table that is visible from all other query clauses and can interact with other tables named in the `from` clause. Here's a simple example:

```
mysql> SELECT e.emp_id, e.fname, e.lname
    -> FROM (SELECT emp_id, fname, lname, start_date, title
    ->       FROM employee) e;
+--------+----------+-----------+
| emp_id | fname    | lname     |
+--------+----------+-----------+
|      1 | Michael  | Smith     |
|      2 | Susan    | Barker    |
|      3 | Robert   | Tyler     |
|      4 | Susan    | Hawthorne |
|      5 | John     | Gooding   |
|      6 | Helen    | Fleming   |
|      7 | Chris    | Tucker    |
|      8 | Sarah    | Parker    |
|      9 | Jane     | Grossman  |
|     10 | Paula    | Roberts   |
|     11 | Thomas   | Ziegler   |
|     12 | Samantha | Jameson   |
|     13 | John     | Blake     |
|     14 | Cindy    | Mason     |
|     15 | Frank    | Portman   |
|     16 | Theresa  | Markham   |
|     17 | Beth     | Fowler    |
|     18 | Rick     | Tulman    |
+--------+----------+-----------+
18 rows in set (0.00 sec)
```

In this example, a subquery against the employee table returns five columns, and the *containing query* references three of the five available columns. The subquery is referenced by the containing query via its alias, which, in this case, is e. This is a simplistic and not particularly useful example of a subquery in a from clause; you will find detailed coverage of subqueries in Chapter 9.

Views

A view is a query that is stored in the data dictionary. It looks and acts like a table, but there is no data associated with a view (this is why I call it a *virtual* table). When you issue a query against a view, your query is merged with the view definition to create a final query to be executed.

To demonstrate, here's a view definition that queries the employee table and includes a call to a built-in function:

```
mysql> CREATE VIEW employee_vw AS
    -> SELECT emp_id, fname, lname,
    ->   YEAR(start_date) start_year
    -> FROM employee;
Query OK, 0 rows affected (0.10 sec)
```

When the view is created, no additional data is generated or stored: the server simply tucks away the select statement for future use. Now that the view exists, you can issue queries against it, as in:

```
mysql> SELECT emp_id, start_year
    -> FROM employee_vw;
+--------+------------+
| emp_id | start_year |
+--------+------------+
|      1 |       2005 |
|      2 |       2006 |
|      3 |       2005 |
|      4 |       2006 |
|      5 |       2007 |
|      6 |       2008 |
|      7 |       2008 |
|      8 |       2006 |
|      9 |       2006 |
|     10 |       2006 |
|     11 |       2004 |
|     12 |       2007 |
|     13 |       2004 |
|     14 |       2006 |
|     15 |       2007 |
|     16 |       2005 |
|     17 |       2006 |
|     18 |       2006 |
+--------+------------+
18 rows in set (0.07 sec)
```

Views are created for various reasons, including to hide columns from users and to simplify complex database designs.

Table Links

The second deviation from the simple `from` clause definition is the mandate that if more than one table appears in the `from` clause, the conditions used to *link* the tables must be included as well. This is not a requirement of MySQL or any other database server, but it is the ANSI-approved method of joining multiple tables, and it is the most portable across the various database servers. We explore joining multiple tables in depth in Chapters 5 and 10, but here's a simple example in case I have piqued your curiosity:

```
mysql> SELECT employee.emp_id, employee.fname,
    ->   employee.lname, department.name dept_name
    -> FROM employee INNER JOIN department
    ->   ON employee.dept_id = department.dept_id;
+--------+----------+-----------+----------------+
| emp_id | fname    | lname     | dept_name      |
+--------+----------+-----------+----------------+
|      1 | Michael  | Smith     | Administration |
|      2 | Susan    | Barker    | Administration |
|      3 | Robert   | Tyler     | Administration |
|      4 | Susan    | Hawthorne | Operations     |
|      5 | John     | Gooding   | Loans          |
|      6 | Helen    | Fleming   | Operations     |
|      7 | Chris    | Tucker    | Operations     |
|      8 | Sarah    | Parker    | Operations     |
|      9 | Jane     | Grossman  | Operations     |
|     10 | Paula    | Roberts   | Operations     |
|     11 | Thomas   | Ziegler   | Operations     |
|     12 | Samantha | Jameson   | Operations     |
|     13 | John     | Blake     | Operations     |
|     14 | Cindy    | Mason     | Operations     |
|     15 | Frank    | Portman   | Operations     |
|     16 | Theresa  | Markham   | Operations     |
|     17 | Beth     | Fowler    | Operations     |
|     18 | Rick     | Tulman    | Operations     |
+--------+----------+-----------+----------------+
18 rows in set (0.05 sec)
```

The previous query displays data from both the `employee` table (`emp_id`, `fname`, `lname`) and the `department` table (`name`), so both tables are included in the `from` clause. The mechanism for linking the two tables (referred to as a *join*) is the employee's department affiliation stored in the `employee` table. Thus, the database server is instructed to use the value of the `dept_id` column in the `employee` table to look up the associated department name in the `department` table. Join conditions for two tables are found in the `on` subclause of the `from` clause; in this case, the join condition is `ON employee.dept_id = department.dept_id`. Again, please refer to Chapter 5 for a thorough discussion of joining multiple tables.

Defining Table Aliases

When multiple tables are joined in a single query, you need a way to identify which table you are referring to when you reference columns in the `select`, `where`, `group by`, `having`, and `order by` clauses. You have two choices when referencing a table outside the `from` clause:

- Use the entire table name, such as `employee.emp_id`.
- Assign each table an *alias* and use the alias throughout the query.

In the previous query, I chose to use the entire table name in the `select` and `on` clauses. Here's what the same query looks like using table aliases:

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee e INNER JOIN department d
  ON e.dept_id = d.dept_id;
```

If you look closely at the `from` clause, you will see that the `employee` table is assigned the alias `e`, and the `department` table is assigned the alias `d`. These aliases are then used in the `on` clause when defining the join condition as well as in the `select` clause when specifying the columns to include in the result set. I hope you will agree that using aliases makes for a more compact statement without causing confusion (as long as your choices for alias names are reasonable). Additionally, you may use the `as` keyword with your table aliases, similar to what was demonstrated earlier for column aliases:

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee AS e INNER JOIN department AS d
  ON e.dept_id = d.dept_id;
```

I have found that roughly half of the database developers I have worked with use the `as` keyword with their column and table aliases, and half do not.

The where Clause

The queries shown thus far in the chapter have selected every row from the `employee`, `department`, or `account` table (except for the demonstration of `distinct` earlier in the chapter). Most of the time, however, you will not wish to retrieve *every* row from a table but will want a way to filter out those rows that are not of interest. This is a job for the `where` clause.

The `where` *clause is the mechanism for filtering out unwanted rows from your result set.*

For example, perhaps you are interested in retrieving data from the `employee` table, but only for those employees who are employed as head tellers. The following query employs a `where` clause to retrieve *only* the four head tellers:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller';
+--------+---------+---------+------------+-------------+
| emp_id | fname   | lname   | start_date | title       |
+--------+---------+---------+------------+-------------+
|      6 | Helen   | Fleming | 2008-03-17 | Head Teller |
|     10 | Paula   | Roberts | 2006-07-27 | Head Teller |
|     13 | John    | Blake   | 2004-05-11 | Head Teller |
|     16 | Theresa | Markham | 2005-03-15 | Head Teller |
+--------+---------+---------+------------+-------------+
4 rows in set (1.17 sec)
```

In this case the `where` clause filtered out 14 of the 18 employee rows. This `where` clause contains a single *filter condition*, but you can include as many conditions as required; individual conditions are separated using operators such as `and`, `or`, and `not` (see Chapter 4 for a complete discussion of the `where` clause and filter conditions). Here's an extension of the previous query that includes a second condition stating that only those employees with a start date later than January 1, 2006 should be included:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   AND start_date > '2006-01-01';
+--------+-------+---------+------------+-------------+
| emp_id | fname | lname   | start_date | title       |
+--------+-------+---------+------------+-------------+
|      6 | Helen | Fleming | 2008-03-17 | Head Teller |
|     10 | Paula | Roberts | 2006-07-27 | Head Teller |
+--------+-------+---------+------------+-------------+
2 rows in set (0.01 sec)
```

The first condition (`title = 'Head Teller'`) filtered out 14 of 18 employee rows, and the second condition (`start_date > '2006-01-01'`) filtered out an additional 2 rows, leaving 2 rows in the final result set. Let's see what would happen if you change the operator separating the two conditions from `and` to `or`:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   OR start_date > '2006-01-01';
+--------+----------+-----------+------------+--------------------+
| emp_id | fname    | lname     | start_date | title              |
+--------+----------+-----------+------------+--------------------+
|      2 | Susan    | Barker    | 2006-09-12 | Vice President     |
|      4 | Susan    | Hawthorne | 2006-04-24 | Operations Manager |
|      5 | John     | Gooding   | 2007-11-14 | Loan Manager       |
|      6 | Helen    | Fleming   | 2008-03-17 | Head Teller        |
|      7 | Chris    | Tucker    | 2008-09-15 | Teller             |
|      8 | Sarah    | Parker    | 2006-12-02 | Teller             |
|      9 | Jane     | Grossman  | 2006-05-03 | Teller             |
|     10 | Paula    | Roberts   | 2006-07-27 | Head Teller        |
|     12 | Samantha | Jameson   | 2007-01-08 | Teller             |
|     13 | John     | Blake     | 2004-05-11 | Head Teller        |
|     14 | Cindy    | Mason     | 2006-08-09 | Teller             |
|     15 | Frank    | Portman   | 2007-04-01 | Teller             |
|     16 | Theresa  | Markham   | 2005-03-15 | Head Teller        |
|     17 | Beth     | Fowler    | 2006-06-29 | Teller             |
|     18 | Rick     | Tulman    | 2006-12-12 | Teller             |
+--------+----------+-----------+------------+--------------------+
15 rows in set (0.00 sec)
```

Looking at the output, you can see that all four head tellers are included in the result set, along with any other employee who started working for the bank after January 1, 2006. At least one of the two conditions is true for 15 of the 18 employees in the `employee` table. Thus, when you separate conditions using the `and` operator, *all* conditions must evaluate to `true` to be included in the result set; when you use `or`, however, only *one* of the conditions needs to evaluate to `true` for a row to be included.

So, what should you do if you need to use both `and` and `or` operators in your `where` clause? Glad you asked. You should use parentheses to group conditions together. The next query specifies that only those employees who are head tellers *and* began working for the company after January 1, 2006 *or* those employees who are tellers *and* began working after January 1, 2007 be included in the result set:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE (title = 'Head Teller' AND start_date > '2006-01-01')
    ->   OR (title = 'Teller' AND start_date > '2007-01-01');
+--------+----------+---------+------------+-------------+
| emp_id | fname    | lname   | start_date | title       |
+--------+----------+---------+------------+-------------+
|      6 | Helen    | Fleming | 2008-03-17 | Head Teller |
|      7 | Chris    | Tucker  | 2008-09-15 | Teller      |
|     10 | Paula    | Roberts | 2006-07-27 | Head Teller |
|     12 | Samantha | Jameson | 2007-01-08 | Teller      |
|     15 | Frank    | Portman | 2007-04-01 | Teller      |
+--------+----------+---------+------------+-------------+
5 rows in set (0.00 sec)
```

You should always use parentheses to separate groups of conditions when mixing different operators so that you, the database server, and anyone who comes along later to modify your code will be on the same page.

The group by and having Clauses

All the queries thus far have retrieved raw data without any manipulation. Sometimes, however, you will want to find trends in your data that will require the database server to cook the data a bit before you retrieve your result set. One such mechanism is the `group by` clause, which is used to group data by column values. For example, rather than looking at a list of employees and the departments to which they are assigned, you might want to look at a list of departments along with the number of employees assigned to each department. When using the `group by` clause, you may also use the `having` clause, which allows you to filter group data in the same way the where clause lets you filter raw data.

Here's a quick look at a query that counts all the employees in each department and returns the names of those departments having more than two employees:

```
mysql> SELECT d.name, count(e.emp_id) num_employees
    -> FROM department d INNER JOIN employee e
    ->   ON d.dept_id = e.dept_id
    -> GROUP BY d.name
    -> HAVING count(e.emp_id) > 2;
+----------------+---------------+
| name           | num_employees |
+----------------+---------------+
| Administration |             3 |
| Operations     |            14 |
+----------------+---------------+
2 rows in set (0.00 sec)
```

I wanted to briefly mention these two clauses so that they don't catch you by surprise later in the book, but they are a bit more advanced than the other four select clauses. Therefore, I ask that you wait until Chapter 8 for a full description of how and when to use group by and having.

The order by Clause

In general, the rows in a result set returned from a query are not in any particular order. If you want your result set in a particular order, you will need to instruct the server to sort the results using the order by clause:

The order by *clause is the mechanism for sorting your result set using either raw column data or expressions based on column data.*

For example, here's another look at an earlier query against the account table:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          13 | CHK        |
|          13 | MM         |
|           1 | CHK        |
|           1 | SAV        |
|           1 | MM         |
|          16 | CHK        |
|           1 | CHK        |
|           1 | CD         |
|          10 | CD         |
|          16 | CHK        |
|          16 | SAV        |
|           1 | CHK        |
|           1 | MM         |
|           1 | CD         |
|          16 | CHK        |
|          16 | BUS        |
|          10 | BUS        |
|          16 | CHK        |
|          13 | SBL        |
+-------------+------------+
24 rows in set (0.00 sec)
```

If you are trying to analyze data for each employee, it would be helpful to sort the results by the `open_emp_id` column; to do so, simply add this column to the `order by` clause:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|           1 | CHK        |
|           1 | SAV        |
|           1 | MM         |
|           1 | CHK        |
|           1 | CD         |
|           1 | CHK        |
|           1 | MM         |
|           1 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | BUS        |
|          13 | CHK        |
|          13 | MM         |
|          13 | SBL        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | SAV        |
|          16 | CHK        |
|          16 | BUS        |
|          16 | CHK        |
+-------------+------------+
24 rows in set (0.00 sec)
```

It is now easier to see what types of accounts each employee opened. However, it might be even better if you could ensure that the account types were shown in the same order for each distinct employee; you can accomplish this by adding the `product_cd` column after the `open_emp_id` column in the `order by` clause:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id, product_cd;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|           1 | CD         |
|           1 | CD         |
|           1 | CHK        |
|           1 | CHK        |
|           1 | CHK        |
|           1 | MM         |
|           1 | MM         |
|           1 | SAV        |
|          10 | BUS        |
|          10 | CD         |
|          10 | CD         |
|          10 | CHK        |
|          10 | CHK        |
|          10 | SAV        |
|          10 | SAV        |
|          13 | CHK        |
|          13 | MM         |
|          13 | SBL        |
|          16 | BUS        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | SAV        |
+-------------+------------+
24 rows in set (0.00 sec)
```

The result set has now been sorted first by employee ID and then by account type. The order in which columns appear in your `order by` clause does make a difference.

Ascending Versus Descending Sort Order

When sorting, you have the option of specifying *ascending* or *descending* order via the `asc` and `desc` keywords. The default is ascending, so you will need to add the `desc` keyword, only if you want to use a descending sort. For example, the following query lists all accounts sorted by available balance with the highest balance listed at the top:

```
mysql> SELECT account_id, product_cd, open_date, avail_balance
    -> FROM account
    -> ORDER BY avail_balance DESC;
+------------+------------+------------+---------------+
| account_id | product_cd | open_date  | avail_balance |
+------------+------------+------------+---------------+
|         29 | SBL        | 2004-02-22 |      50000.00 |
|         28 | CHK        | 2003-07-30 |      38552.05 |
|         24 | CHK        | 2002-09-30 |      23575.12 |
|         15 | CD         | 2004-12-28 |      10000.00 |
|         27 | BUS        | 2004-03-22 |       9345.55 |
|         22 | MM         | 2004-10-28 |        9345.55 |
|         12 | MM         | 2004-09-30 |        5487.09 |
|         17 | CD         | 2004-01-12 |        5000.00 |
|         18 | CHK        | 2001-05-23 |        3487.19 |
|          3 | CD         | 2004-06-30 |        3000.00 |
|          4 | CHK        | 2001-03-12 |        2258.02 |
|         13 | CHK        | 2004-01-27 |        2237.97 |
|          8 | MM         | 2002-12-15 |        2212.50 |
|         23 | CD         | 2004-06-30 |        1500.00 |
|          1 | CHK        | 2000-01-15 |        1057.75 |
|          7 | CHK        | 2002-11-23 |        1057.75 |
|         11 | SAV        | 2000-01-15 |         767.77 |
|         10 | CHK        | 2003-09-12 |         534.12 |
|          2 | SAV        | 2000-01-15 |         500.00 |
|         19 | SAV        | 2001-05-23 |         387.99 |
|          5 | SAV        | 2001-03-12 |         200.00 |
|         21 | CHK        | 2003-07-30 |         125.67 |
|         14 | CHK        | 2002-08-24 |         122.37 |
|         25 | BUS        | 2002-10-01 |           0.00 |
+------------+------------+------------+----------------+
24 rows in set (0.05 sec)
```

Descending sorts are commonly used for ranking queries, such as “show me the top five account balances.” MySQL includes a `limit` clause that allows you to sort your data and then discard all but the first *X* rows; see Appendix B for a discussion of the `limit` clause, along with other non-ANSI extensions.

Sorting via Expressions

Sorting your results using column data is all well and good, but sometimes you might need to sort by something that is not stored in the database, and possibly doesn’t appear anywhere in your query. You can add an expression to your `order by` clause to handle such situations. For example, perhaps you would like to sort your customer data by the last three digits of the customer’s federal ID number (which is either a Social Security number for individuals or a corporate ID for businesses):

```
mysql> SELECT cust_id, cust_type_cd, city, state, fed_id
    -> FROM customer
    -> ORDER BY RIGHT(fed_id, 3);
+---------+-------------+------------+-------+-------------+
| cust_id | cust_type_cd | city       | state | fed_id      |
+---------+-------------+------------+-------+-------------+
|       1 | I           | Lynnfield  | MA    | 111-11-1111 |
|      10 | B           | Salem      | NH    | 04-1111111  |
|       2 | I           | Woburn     | MA    | 222-22-2222 |
|      11 | B           | Wilmington | MA    | 04-2222222  |
|       3 | I           | Quincy     | MA    | 333-33-3333 |
|      12 | B           | Salem      | NH    | 04-3333333  |
|      13 | B           | Quincy     | MA    | 04-4444444  |
|       4 | I           | Waltham    | MA    | 444-44-4444 |
|       5 | I           | Salem      | NH    | 555-55-5555 |
|       6 | I           | Waltham    | MA    | 666-66-6666 |
|       7 | I           | Wilmington | MA    | 777-77-7777 |
|       8 | I           | Salem      | NH    | 888-88-8888 |
|       9 | I           | Newton     | MA    | 999-99-9999 |
+---------+-------------+-----------+-------+-------------+
13 rows in set (0.24 sec)
```

This query uses the built-in function `right()` to extract the last three characters of the `fed_id` column and then sorts the rows based on this value.

Sorting via Numeric Placeholders

If you are sorting using the columns in your `select` clause, you can opt to reference the columns by their *position* in the `select` clause rather than by name. For example, if you want to sort using the second and fifth columns returned by a query, you could do the following:

```
mysql> SELECT emp_id, title, start_date, fname, lname
    -> FROM employee
    -> ORDER BY 2, 5;
+--------+--------------------+------------+----------+-----------+
| emp_id | title              | start_date | fname    | lname     |
+--------+--------------------+------------+----------+-----------+
|     13 | Head Teller        | 2004-05-11 | John     | Blake     |
|      6 | Head Teller        | 2008-03-17 | Helen    | Fleming   |
|     16 | Head Teller        | 2005-03-15 | Theresa  | Markham   |
|     10 | Head Teller        | 2006-07-27 | Paula    | Roberts   |
|      5 | Loan Manager       | 2007-11-14 | John     | Gooding   |
|      4 | Operations Manager | 2006-04-24 | Susan    | Hawthorne |
|      1 | President          | 2005-06-22 | Michael  | Smith     |
|     17 | Teller             | 2006-06-29 | Beth     | Fowler    |
|      9 | Teller             | 2006-05-03 | Jane     | Grossman  |
|     12 | Teller             | 2007-01-08 | Samantha | Jameson   |
|     14 | Teller             | 2006-08-09 | Cindy    | Mason     |
|      8 | Teller             | 2006-12-02 | Sarah    | Parker    |
|     15 | Teller             | 2007-04-01 | Frank    | Portman   |
|      7 | Teller             | 2008-09-15 | Chris    | Tucker    |
|     18 | Teller             | 2006-12-12 | Rick     | Tulman    |
|     11 | Teller             | 2004-10-23 | Thomas   | Ziegler   |
|      3 | Treasurer          | 2005-02-09 | Robert   | Tyler     |
|      2 | Vice President     | 2006-09-12 | Susan    | Barker    |
+--------+--------------------+------------+----------+-----------+
18 rows in set (0.00 sec)
```

You might want to use this feature sparingly, since adding a column to the `select` clause without changing the numbers in the `order by` clause can lead to unexpected results. Personally, I may reference columns positionally when writing ad hoc queries, but I always reference columns by name when writing code.