15

Optimizing Performance

In many examples throughout this book, you have seen SQL statements executed against small tables that contain relatively few rows. As a result, the performance of these statements has not been an issue because it takes relatively little time for MySQL to return information or modify data. This is often not the case, however, in the real world. If you're accessing tables that contain thousands of rows of data (or more), you might find that certain SQL statements are slow and take a relatively long time to be processed, despite how efficiently you think that statement should run. As a result, whenever you're setting up a database or creating SQL statements to execute against the database, you should take into consideration how well those statements perform when they are executed.

When you begin working with tables that contain large quantities of data, there are several steps that you can take to optimize the performance of your SQL statements. By optimizing performance, you're maximizing the speed and efficiency at which those statements are executed. For example, in order to ensure that your `SELECT` statements retrieve data as quickly as possible, you can ensure that your tables have been properly indexed. In this chapter, you learn about various steps that you can take to optimize your system's performance. Specifically, the chapter covers the following topics:

- Advantages and disadvantages of indexing and when you should use indexing
- Determining how effectively your queries are being executed and steps you can take to improve your data-related operations
- Modifying table definitions to improve query performance
- Enabling your system's query cache

Optimizing MySQL Indexing

In MySQL, the most useful step that you can take to maximize performance is to ensure that your tables are properly indexed. Indexes provide an effective way to access data in your tables and speed up searches. An index provides an organized list of pointers to the actual data. As a result, when MySQL executes a query, it can scan the index to locate the correct data, rather than having to scan the entire table.

Figure 15-1 helps to illustrate how this works. The figure shows a table named Parts, which includes the PartID column, the PartName column, and the ManfID column. The Parts table, as with any table, is basically a collection of rows. Although in this case the rows are ordered by the PartID values, they can be in any order.

Parts

PartID	PartName	ManfID
101	DVD burner	abc123
102	CD drive	jkl123
103	80-GB hard disk	mno456
104	Mini-tower	ghi789
105	Power supply	def456
106	LCD monitor	mno456
107	Zip drive	ghi789
109	Floppy drive	jkl123
109	Network adapter	def456
110	Network hub	jkl123
111	Router	mno456
112	Sound card	ghi789
113	Standard keyboard	mno456
114	PS/2 mouse	jkl123
115	56-K modem	ghi789
116	Display adapter	mno456
117	IDE controller	def456

Figure 15-1

Assume for now that no indexes have been defined on the Parts table. Now imagine that you want to run the following `SELECT` statement:

```
SELECT PartName FROM Parts
WHERE ManfID='jkl123';
```

When you execute the statement, MySQL must search through each row in the Parts table to find those rows that have a ManfID value of jkl123. If the table includes a large number of rows, this process can be slow and very inefficient. Now take a look at Figure 15-2, which shows that same table, only this time an index is defined on the ManfID column. The index contains exactly the same values as the ManfID column; however, the values are sorted in ascending order. In addition, each value in the index contains a pointer to the applicable row in the Parts column.

Now when you execute the `SELECT` statement, MySQL searches the index, rather than searching the entire table row by row and, as a result, can find the ManfID value of jkl123 much faster. Searches are faster in indexes because indexes are sorted. Identical values are grouped together and organized in an easy-to-locate order. In addition, because of this sorting, MySQL knows when to stop searching. As soon as it reaches the end of the matching rows, it discontinues the search. For example, if you refer to the index in Figure 15-2, you can see that all the jkl123 values are grouped together. This means that they can all be located with one search and that the last one is easy to identify. The process is made even more efficient because MySQL uses a special positioning algorithm that locates the first matching entry, without having to start the scan at the beginning of the index.

Parts

PartID	PartName	ManfID
101	DVD burner	abc123
102	CD drive	jkl123
103	80-GB hard disk	mno456
104	Mini-tower	ghi789
105	Power supply	def456
106	LCD monitor	mno456
107	Zip drive	ghi789
109	Floppy drive	jkl123
109	Network adapter	def456
110	Network hub	jkl123
111	Router	mno456
112	Sound card	ghi789
113	Standard keyboard	mno456
114	PS/2 mouse	jkl123
115	56-K modem	ghi789
116	Display adapter	mno456
117	IDE controller	def456

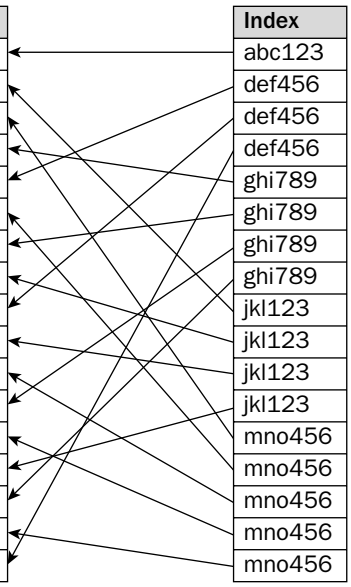

Figure 15-2

The benefit of indexes becomes even more apparent for queries that join multiple tables. As you recall from Chapter 10, when you join two or more tables, rows in each table are matched together based on the specified join condition. If the tables are not indexed, MySQL must compare each row in each table with each row in each other table to determine which rows contain the values that meet the join criteria. This means that MySQL must try every possible combination in the joined tables to determine which rows meet the join criteria.

For example, suppose you execute a `SELECT` statement that joins tables A, B, and C, none of which are configured with indexes. Each row in table A is combined with each row in tables B and C to determine whether each combination matches the join condition. In other words, the first row in table A is combined with the first row in table B and the first row in table C. The first row in table A is then combined with the first row in table B and the second row in table C. The first row in table A is then combined with the first row in table B and the third row in table C. The process is repeated until each row in table C is covered. The process begins again as the first row in table A is combined with the second row in table B and the first row in table C. The first row in table A is then combined with the second row in table B and the second row in table C. This continues until all possible combinations are compared, which can result in excessively large searches. For example, if each table contains 100 rows, MySQL must compare one million rows (100 x 100 x 100) to determine which rows match the join condition.

Indexes, though, eliminate the need to compare all the rows in all the tables. Returning to tables A, B, and C, assume that the columns specified in the join condition are now indexed. When you execute your `SELECT` statement, MySQL uses an index to locate the first applicable row in table A. MySQL then uses an index on table B to match the row in table A to the appropriate row in table B. From there, MySQL uses a table C index to match the row in table B to the appropriate row in table C. Each step of the way, MySQL uses indexes to locate values in order to match rows according to how the join condition has been defined. As a result, the number of rows that must be searched is significantly reduced, which results in a dramatic improvement in performance.

Despite the clear advantages of using indexes, there are some drawbacks. For example, indexes can require a great deal of storage. The larger the tables and greater the number of indexes, the more storage you need to hold those indexes. You must allow for file size and potential growth whenever you implement an index. Another disadvantage to indexes is that, while they speed up data retrieval, they can slow down data inserts and deletes, as well as updates to columns that are indexed. Any change made to rows in an indexed table must also be made to the index (unless the change is an update that doesn't affect the value in the index).

Despite these drawbacks, indexing provides the most beneficial tool for improving the performance of your `SELECT` statements. You should not, however, index every column in a table. The following list provides several guidelines that you can use in determining when to implement indexing:

- **Index columns that appear in search conditions.** As a general rule, you should consider defining an index on any column that you commonly use in `WHERE`, `GROUP BY`, or `HAVING` clauses. Because these columns define the limitations of a query, they are good candidates for improving performance because they allow MySQL to identify quickly which rows should be included in a search and which should not.
- **Index columns that appear in join conditions.** Index any columns that appear in a join condition. Because join conditions are often based on foreign key columns that reference primary key columns, MySQL creates the indexes automatically when you define the primary keys and foreign keys.
- **Do not index columns that appear only in the `SELECT` clause.** If a column appears in the `SELECT` clause of a `SELECT` statement, but does not appear in `WHERE`, `GROUP BY`, or `HAVING` clauses, you usually shouldn't index these columns because indexing them provides no performance benefit but does require additional storage. Indexing columns in the `SELECT` clause provides no benefit because the `SELECT` clause is one of the last parts of a `SELECT` statement to be processed. MySQL conducts searches based on the other clauses. After MySQL identifies which rows to return, it then consults the `SELECT` clause to determine which columns from the identified rows to return.
- **Do not index columns that contain only a few different values.** If a column contains many duplicated values, indexing that column provides little benefit. For example, suppose that your column is configured to accept only Y and N values. Because of the way in which MySQL accesses an index and uses that index to locate the rows in the tables, many duplicated values can actually cause the process to take longer than if no index is used. In fact, when MySQL finds that a value occurs in more than 30 percent of a table's rows, it usually doesn't use the index at all.
- **Specify prefixes for indexes on columns that contain large string values.** If you're adding an index to a string column, consider defining a prefix on that index so that your index includes only part of the entire values, as they're stored in the table. For example, if your table includes a `CHAR(150)` column, you might consider indexing only the first 10 or 15 bytes, or whatever number provides enough unique values without having to store the entire values in the index.
- **Create only the indexes that you need.** Never create more indexes than you need. If a column is rarely used in a search or join condition, don't index that column. You want to index only those columns that are frequently used to identify the rows being searched.

Once you determine whether to index a column you must decide which type of index to use. In Chapter 5, you learned how to use the `CREATE TABLE` statement to add tables to your MySQL database. As you recall from that chapter, you can define one or more indexes as part of that statement. MySQL supports five types of indexes:

- **Primary key:** Requires that each value or set of values be unique in the columns on which the primary key is defined. In addition, `NULL` values are not allowed. A table can include only one primary key.
- **Foreign key:** Enforces the relationship between the referencing columns in the child table where the foreign key is defined and the referenced columns in the parent table.
- **Regular:** Provides a basic index that permits duplicate values and `NULL` values in the columns on which the index is defined.
- **Unique:** Requires that each value or set of values be unique in the columns on which the index is defined. Unlike primary key indexes, `NULL` values are allowed.
- **Full-text:** Supports full-text searches of the values in the columns on which the index is defined. A full-text index permits duplicate values and `NULL` values in those columns. A full-text index can be defined only on `MyISAM` tables and only on `CHAR`, `VARCHAR`, and `TEXT` columns.

Refer to Chapter 5 for more details about each type of index and how they're defined on a table in a MySQL database.

The type of index you should use depends on your particular requirements. Because primary key and foreign key indexes provide such specific purposes, their use is fairly evident. You should configure every table with one primary key, and you should use foreign keys whenever columns in one table reference columns in another table. In many cases, the use of these keys alone is enough to meet your indexing needs. Should you require additional indexes, you should try to use a unique index over regular and full-text indexes because indexes that contain unique values provide the best performance. MySQL must locate only one value and that value is matched to exactly one row. As soon as MySQL finds that value, the search is complete.

Once you have defined the necessary indexes on your table, you may have addressed many of the performance issues that you experience when running queries against a MySQL database. Even a well-indexed table can experience performance problems. As a result, you should also consider other methods that you can use to optimize your queries.

Optimizing SQL Queries

When you're setting up your database or configuring system settings, you're likely to perform a number of operations such as creating tables, granting privileges to users, or setting the values of system variables. Once your MySQL server and databases have been set up the way you want them, most of the access to the server and database is through applications that retrieve and modify data, which means that statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` represent the most common operations performed. As a result, you should ensure that your system is fully optimized to support these operations. To do so, you must take into account not only methods that you can use to improve data retrieval, but also methods that improve inserting data, updating data, and deleting data.

Optimizing Data Retrieval

Of all the operations performed against a MySQL database, data retrieval operations, through the use of different types of SELECT statements, are the most common. When you execute a SELECT statement, MySQL uses the query optimizer to analyze the statement and perform the query as effectively as possible. The optimizer is a MySQL component whose sole purpose is to ensure the best performance possible for each query by planning out the execution of that query.

The optimizer's primary goal is to try to use indexes whenever possible to process the statement. As you saw earlier in the chapter, an index can provide the most efficient method for locating rows that are accessed by the statement. The optimizer tries to determine which indexes benefit the execution of the statement and which ones don't. In some cases, the optimizer determines that it is better to bypass the index, rather than use it, such as when there are too many duplicate values.

The optimizer also tries to determine the greatest number of rows that can be eliminated from the search. To better understand how this works, take a look at the following SELECT statement:

```
SELECT BookTitle FROM Books
WHERE InStock>20 AND OnOrder>10;
```

Assume that the statement returns 20 rows, with each row meeting the conditions specified by the two expressions in the WHERE clause. When the optimizer first looks at the statement, it tests each of these expressions in order to estimate the number of rows that must be examined to meet the conditions specified by the expressions. You can also assume that, in this case, the optimizer estimates that 600 rows need to be examined for the first expression. If only 20 of the 600 rows meet both search conditions, MySQL has to search through 570 rows that don't meet both search conditions. The optimizer, however, also estimates that only 100 rows have to be examined for the second expression, which means that only 80 rows fail to meet both search conditions.

Based on the results of its initial tests of the two conditions, the query optimizer decides that it is best to process the second expression first and then, from those results, process the first expression. This way, fewer rows have to be processed, which means that the SELECT statement requires less processing time and fewer disk I/O operations than if the first expression is processed first.

The optimizer includes other capabilities that lend to the optimization of a statement; however, despite how efficient the optimizer can be, you might try to execute statements that you believe are not performing as well as they could. As a result, you must sometimes analyze your SELECT statements to determine what steps you can take to improve performance. The most effective method that you can use to analyze your SELECT statement is to use the EXPLAIN statement.

Using the EXPLAIN Statement

The EXPLAIN statement provides an analysis of a specified SELECT statement. To use the EXPLAIN statement, simply include the EXPLAIN keyword, followed by the SELECT statement, as shown in the following syntax:

```
EXPLAIN <select statement>
```

You should include your SELECT statement after the EXPLAIN keyword exactly as you would use the SELECT statement in a query. The EXPLAIN statement then returns results that provide details about how the SELECT statement will be executed. From these details, you can determine whether indexes are being used effectively, whether you should add new indexes, or whether you should specify the order of how tables are joined together.

> *You can also use the* EXPLAIN *statement to return details about a table. To view table details, use the following syntax:* EXPLAIN <table name>. *As you can see, the name of the table follows the* EXPLAIN *keyword. Using the* EXPLAIN *statement in this way produces the same results as using the* DESCRIBE <table name> *statement.*

The best way to understand the EXPLAIN statement is to look at an example. The example is based on a SELECT statement that joins two tables: Manufacturers and Parts. The following table definition describes how the Manufacturers table is created:

```
CREATE TABLE Manufacturers
(
   ManfID CHAR(8)NOT NULL PRIMARY KEY,
   ManfName VARCHAR(30) NOT NULL
)
ENGINE=INNODB;
```

For the purpose of the example, you can assume that the following INSERT statement has populated the Manufacturers table:

```
INSERT INTO Manufacturers
VALUES ('abc123', 'ABC Manufacturing'),
('def456', 'DEF Inc.'),
('ghi789', 'GHI Corporation'),
('jkl123', 'JKL Limited'),
('mno456', 'MNO Company');
```

The following table definition describes how the Parts table is created:

```
CREATE TABLE Parts
(
   PartID SMALLINT NOT NULL PRIMARY KEY,
   PartName VARCHAR(30) NOT NULL,
   ManfID CHAR(8) NOT NULL
)
ENGINE=INNODB;
```

The Parts table shown in this definition is the same Parts table shown in Figure 15-1. As you can see, no foreign key is defined on this table. In reality, you would probably define a foreign key on the ManfID column of the Parts table that references the ManfID column of the Manufacturers table, but it is not done here in order to demonstrate how to analyze a SELECT statement.

Now assume that the following INSERT statement has been used to populate the Parts table:

```
INSERT INTO Parts
VALUES (101, 'DVD burner', 'abc123'),
(102, 'CD drive', 'jkl123'),
(103, '80-GB hard disk', 'mno456'),
(104, 'Mini-tower', 'ghi789'),
(105, 'Power supply', 'def456'),
(106, 'LCD monitor', 'mno456'),
(107, 'Zip drive', 'ghi789'),
(108, 'Floppy drive', 'jkl123'),
(109, 'Network adapter', 'def456'),
(110, 'Network hub', 'jkl123'),
(111, 'Router', 'mno456'),
(112, 'Sound card', 'ghi789'),
(113, 'Standard keyboard', 'mno456'),
(114, 'PS/2 mouse', 'jkl123'),
(115, '56-K modem', 'ghi789'),
(116, 'Display adapter', 'mno456'),
(117, 'IDE controller', 'def456');
```

Once the tables are created and populated, you can execute SELECT statements against the tables in order to retrieve data in those tables. For example, you might execute the following SELECT statement:

```
SELECT PartName, ManfName
FROM Parts AS p, Manufacturers as m
WHERE p.ManfID = m.ManfID
ORDER BY PartName;
```

The SELECT statement returns results similar to the following:

```
+-------------------+-------------------+
| PartName          | ManfName          |
+-------------------+-------------------+
| 56-K modem        | GHI Corporation   |
| 80-GB hard disk   | MNO Company       |
| CD drive          | JKL Limited       |
| Display adapter   | MNO Company       |
| DVD burner        | ABC Manufacturing |
| Floppy drive      | JKL Limited       |
| IDE controller    | DEF Inc.          |
| LCD monitor       | MNO Company       |
| Mini-tower        | GHI Corporation   |
| Network adapter   | DEF Inc.          |
| Network hub       | JKL Limited       |
| Power supply      | DEF Inc.          |
| PS/2 mouse        | JKL Limited       |
| Router            | MNO Company       |
| Sound card        | GHI Corporation   |
| Standard keyboard | MNO Company       |
| Zip drive         | GHI Corporation   |
+-------------------+-------------------+
17 rows in set (0.12 sec)
```

After executing the statement, you might decide that the statement's performance could be better. Although this might not be apparent when each table contains so few rows, it would become so if the tables contained thousands of rows. As a result, you decide to analyze this statement to see whether any bottlenecks exist and, if so, where they might be. To analyze the SELECT statement, you simply precede it with the EXPLAIN keyword, as shown in the following statement:

```
EXPLAIN SELECT PartName, ManfName
FROM Parts AS p, Manufacturers as m
WHERE p.ManfID = m.ManfID
ORDER BY PartName;
```

When you execute this statement, you should receive results similar to the following:

```
+----+-------------+-------+------+---------------+------+---------+------+------+-
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows |
+----+-------------+-------+------+---------------+------+---------+------+------+-
|  1 | SIMPLE      | p     | ALL  | NULL          | NULL |    NULL | NULL |   17 |
|  1 | SIMPLE      | m     | ALL  | PRIMARY       | NULL |    NULL | NULL |    4 |
+----+-------------+-------+------+---------------+------+---------+------+------+-
2 rows in set (0.00 sec)
```

The results returned by the EXPLAIN statement include a row for each table that participates in the query. The results shown here represent only a part of the entire results. The results that you can expect also include a column named Extra. In the preceding statement, the Extra column displays the value `Using temporary; Using filesort` for the first row and displays the value `Using where` in the second row.

Each column returned by the EXPLAIN statement provides specific information about how the optimizer plans to execute the SELECT statement. The following table describes each column returned by the EXPLAIN statement and provides information about the values displayed in the previous results set.

Column	Description
id	An identifier for the SELECT statement that is being analyzed. If the statement doesn't include subqueries or unions, the id value is 1 for each row, as is the case in the preceding example results set.
select_type	The type of SELECT statement. The SIMPLE value indicates that the statement doesn't include subqueries or unions. Other values indicate how the statement participates in a subquery or union.
table	The table being analyzed by the row. If an alias is used for the table name, the column displays the alias, rather than the actual table name.
type	The method used to match rows in different tables when the SELECT statement joins two or more tables. If ALL is specified, MySQL conducts a full table scan for each combination of rows from the current table and the joined table. Generally, you should avoid ALL in all but the first row.
possible_keys	The indexes that MySQL can use to find rows. If NULL, no indexes can be used. In the previous example, the primary key in the Manufacturers table can potentially be used to process the SELECT statement. This index would be considered because it is one of the columns specified in the join condition.

Table continued on following page

Column	Description
key	The indexes that MySQL actually uses to return rows. If `NULL`, no indexes are used.
key_len	The length of the index used to retrieve rows. This is most useful in determining how many parts of a multicolumn index are used. For example, if an index is made up of two columns that are each configured as `CHAR(4)` columns and the key_len column value is 4, you know that only the first column in the index is used. The key_len value is `NULL` if the key value is `NULL`.
ref	The column used in conjunction with the index specified in the key column. This usually refers to the column referenced in a foreign key. If `NULL`, no columns are used.
rows	The number of rows that MySQL plans to examine in order to execute the query. This column is normally your best indicator of the efficiency of the column. The more rows that must be examined, the less efficient the query.
Extra	Additional information about the query. For example, if a query can be executed by referring only to the index, the value `Using index` is displayed. The `Using filesort` value is displayed if MySQL must make an additional pass to retrieve rows in a sorted order. The `Using temporary` value is displayed if MySQL will create a temporary table to execute the query. The `Using where` value indicates that the `WHERE` clause will be used to restrict which rows to retrieve.

This table includes only the basic information that you need to understand the results returned by an `EXPLAIN` *statement. For more information about the results returned by the statement, see the MySQL product documentation.*

As the `EXPLAIN` statement results in the preceding example indicate, the `SELECT` statement will be processed by examining 17 rows in the Parts table and 4 rows in the Manufacturers table. In addition, because `ALL` is specified as the scan type, you know that a full-table scan will be conducted on both tables. To arrive at how many rows will actually be examined, you should multiply the values in the rows column. In this case, 68 rows will be examined. As you'll recall from the query results for this statement, only 17 rows are actually returned. As the `EXPLAIN` statement results also show, no indexes are being used to process the `SELECT` statement.

The first step that you should take is to try to determine whether you should add any more indexes to the table. As you would expect, the place to start is to add a foreign key to the Parts table. The foreign key not only provides referential integrity but creates an index on that column, which is important because that column is specified in the join condition. You can add the foreign key by using the following `ALTER TABLE` statement:

```
ALTER TABLE Parts
ADD FOREIGN KEY (ManfID) REFERENCES Manufacturers (ManfID);
```

After you modify the Parts table, you can use the same EXPLAIN statement that you saw in the preceding example. You should now receive results similar to the following:

```
+----+-------------+-------+------+---------------+------+---------+------+------+-
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows |
+----+-------------+-------+------+---------------+------+---------+------+------+-
|  1 | SIMPLE      | m     | ALL  | PRIMARY       | NULL |    NULL | NULL |    5 |
|  1 | SIMPLE      | p     | ALL  | ManfID        | NULL |    NULL | NULL |   13 |
+----+-------------+-------+------+---------------+------+---------+------+------+-
2 rows in set (0.00 sec)
```

The results shown here are similar to the results displayed the first time you executed the EXPLAIN statement. However, the new results show that 5 rows in the Manufacturers table and 13 in the Parts table will be scanned. When you multiply these values, you find that 65 rows will be examined to execute this query, not a very big improvement over the last time. One reason for this is that, although the ManfID index is listed in the possible_keys column, it is not being used. As a result, MySQL will still do a full-table scan on both tables.

The SELECT statement has not seen a great improvement in performance because the cardinality of the index is improperly set when you use the ALTER TABLE statement to create that index. Index *cardinality* refers to the number of unique values in an index. For example, suppose that you have a column that permits only three values. If you create an index on the column, the cardinality for that index is 3, no matter how many rows are in that table. In general, MySQL does not use an index with a low cardinality because this is not an efficient use of indexes. As a result, when you add an index to a table or modify it significantly in any other way, you should ensure that the cardinality is correctly represented to the query optimizer. The easiest way to do this is to execute an OPTIMIZE TABLE statement.

> *You can view a table's cardinality by using the* SHOW INDEX *statement on that table. Viewing the cardinality does not necessarily tell you whether that setting is improperly set, so you still might want to run the* OPTIMIZE TABLE *statement.*

Using the OPTIMIZE TABLE Statement

The OPTIMIZE TABLE statement performs a number of functions. For example, it defragments the table and sorts the table's indexes. It also updates the internal table statistics. One of these statistics is the cardinality of its indexes. If you add an index to an existing table, you might have to use the OPTIMIZE TABLE statement to ensure that the table statistics are accurate when read by the query optimizer. The following syntax shows how to create an OPTIMIZE TABLE statement:

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE <table name> [{, <table name>}...]
```

As the syntax shows, you must, at a minimum, specify the OPTIMIZE TABLE keywords and the name of the table. In addition, you can specify the LOCAL option or the NO_WRITE_TO_BIN_LOG option (which are synonymous) to prevent the optimization process from being written to the binary log. In addition, you can specify more than one table.

Now return to the examples from the previous section. Because you added an index to the Parts table, you should run the OPTIMIZE TABLE statement on that table, as shown in the following example:

```
OPTIMIZE TABLE Parts;
```

When you execute this statement, you should receive results similar to the following:

```
+------------+----------+----------+----------+
| Table      | Op       | Msg_type | Msg_text |
+------------+----------+----------+----------+
| test.parts | optimize | status   | OK       |
+------------+----------+----------+----------+
1 row in set (0.35 sec)
```

Basically, these results are telling you that the parts table has been optimized. Now when you run an `EXPLAIN` statement (against the same example `SELECT` statement used previously), your results should be similar to the following:

```
------+------+---------------+--------+---------+---------------+------+-----------
table | type | possible_keys | key    | key_len | ref           | rows | Extra
------+------+---------------+--------+---------+---------------+------+-----------
m     | ALL  | PRIMARY       | NULL   |    NULL | NULL          |    5 | Using temp
p     | ref  | ManfID        | ManfID |       8 | test.m.ManfID |    1 |
------+------+---------------+--------+---------+---------------+------+-----------
2 rows in set (0.00 sec)
```

Again, these results show only a part of the entire results. The id and select_type columns are not included, but they show the same results as in the earlier examples. The id value for both rows is 1, and the select_type value for both rows is `SIMPLE`. In addition, only part of the Extra column is shown in these results. The entire Extra value for the row about the Manufacturers table is `Using temporary; Using filesort`, indicating that a temporary table is used in processing the `SELECT` statement and an additional pass is made to retrieve the rows in a sorted order.

Although the id and select_type values are the same as in the previous example, there are a number of differences. First, the type column for the Parts table shows a value of `ref`, rather than `ALL`, indicating that the rows from this table are read based on index values that are matched according to the join condition, rather than performing a full-table scan. In addition, the optimizer now uses the index defined on the ManfID column of the Parts table, in conjunction with the ManfID column of the Manufacturers table, as shown in the ref column. The most important statistics in these results are in the rows column, which shows that 5 rows will be searched in the Manufacturers table and 1 row searched in the Parts table, indicating that only 5 rows will be examined to process this query, rather than 65.

At this point, you may wonder how it's possible that so few rows can be examined when you know that 17 rows are returned by the `SELECT` statement. The problem is that the values that are shown in the rows column are only estimates of the number of rows that the optimizer believes must be processed in order to execute the query. Because of the method the optimizer uses to arrive at these estimates, the smaller the table, the more inaccurate the estimates can be. And in reality, if you are working with tables as small as the ones shown in these examples, you do not need to be too concerned about optimizing your queries. As your tables grow and they contain thousands of rows, or even millions, optimization becomes critical. Regardless of the exact amount shown in the rows column, your goal should still be to get that total row count down as low as possible, and the steps shown here are a good way to start.

Adding indexes and executing `OPTIMIZE TABLE` statements are not the only methods that you can use to optimize query performance. MySQL also recommends other steps that you can take to maximize performance.

Understanding the SELECT Statement Guidelines

As stated earlier, proper indexing should still be your first strategy in optimizing your system so that you can retrieve data as efficiently as possible. There are several other steps that you can take to improve performance, as described in the following guidelines:

- **Do not use unnecessary wildcards in `LIKE` clauses.** Use wildcards only when you need them. For example, if you are looking for values that begin with "Cha," don't specify a wildcard at the beginning of your value, as in `LIKE '%cha%'`. Instead, omit the first wildcard. (For more information about using the `LIKE` clause, see Chapter 8.)
- **Isolate indexed columns in comparison expressions.** MySQL cannot use an index on a column if that column appears as an argument in a function or an arithmetic expression. For example, suppose your `WHERE` clause includes the expression `YEAR(DateJoined)>1999`, where DateJoined is a column that contains date values. If DateJoined is an indexed column, you might want to rewrite the `WHERE` clause to something similar to the following: `DateJoined>'1999-12-31'`. (For information about using functions, see Chapter 9. For information about expressions, see Chapter 8.)
- **Turn subqueries into joins.** In some cases, you can rewrite a subquery into a join. MySQL processes joins more efficiently than subqueries, so if using a join is an alternative, you should try that. (For more information about joins and subqueries, see Chapter 10.)
- **Try using the `FORCE INDEX` clause.** At times the query optimizer chooses to process a `SELECT` statement without using a particular index. You can try adding the `FORCE INDEX` clause to your `SELECT` statement to force the statement to use the specified index. (You add the `FORCE INDEX` clause to the table reference in your `SELECT` statement. For more information about how to include this option in your `SELECT` statement, see Chapter 7.)
- **Try using the `STRAIGHT_JOIN` option.** When the query optimizer analyzes a `SELECT` statement, it determines the order in which tables will be joined. In some cases, you might find that forcing the optimizer to join tables in the order specified in the `SELECT` statement improves the statement's performance. This occurs because there are times when the query optimizer does not join tables in the most optimal order. As a result, more rows are examined than need to be examined in order to perform an effective join operation. By forcing the join order, you can sometimes see an improvement in performance because fewer rows are being searched. To force the order, you can add the `STRAIGHT_JOIN` option to your `SELECT` statement. (For more information about using the `STRAIGHT_JOIN` option, see Chapter 10.)
- **Try alternative forms of the query.** Sometimes you can improve performance simply by changing how you structure a `SELECT` statement. One example is turning subqueries into joins. Another example is changing the order of tables specified in a join condition. The more you use SQL, the more alternative methods that you'll find to perform the same operations. As a result, it is sometimes worth the effort to try different forms of a `SELECT` statement to produce the same results. You can then analyze each form to determine which version of the statement performs the best. If you decide to try out different versions of a `SELECT` statement, be certain to execute each one several times to ensure that one statement isn't simply reading from the disk cache for the previous statement.

Now that you have a good overview of how to optimize the performance of your `SELECT` statements, you can try out some of what you have learned. In the following exercise, you add two tables to the DVDRentals database. The tables match customers to the cities in which they live. After you create the tables, you execute a `SELECT` statement that retrieves data from the new tables as well as the Customers table. You then use `EXPLAIN` statements to determine what steps you can take to improve the performance of that `SELECT` statement.

Try It Out Optimizing Performance of a SELECT Statement

The following steps describe how to optimize the performance of a `SELECT` statement:

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

```
use DVDRentals
```

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

2. First, create a table for the name of the cities. Execute the following SQL statement at the mysql command prompt:

```
CREATE TABLE Cities
(
   CityID SMALLINT NOT NULL PRIMARY KEY,
   CityName VARCHAR(20) NOT NULL
)
ENGINE=INNODB;
```

   You should receive a message indicating that the table has been successfully created.

3. Now insert data in the table that you created in the previous step. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO Cities
VALUES (101, 'Seattle'), (102, 'Redmond'), (103, 'Bellevue'),
(104, 'Kent'), (105, 'Kirkland');
```

   You should receive a messaging indicating that the statement successfully executed and that five rows were affected.

4. Next, create the second table. Execute the following SQL statement at the mysql command prompt:

```
CREATE TABLE CustCity
(
   CustID SMALLINT NOT NULL,
   CityID SMALLINT NOT NULL
)
ENGINE=INNODB;
```

   You should receive a message indicating that the table has been successfully created.

5. Insert data in the table that you created in Step 4. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO CustCity
VALUES (1, 104), (2, 101), (3, 104),
(4, 103), (5, 102), (6, 105);
```

   You should receive a messaging indicating that the statement successfully executed and that six rows were affected.

6. Before you perform any sort of analysis, you should create a `SELECT` statement that retrieves information from the Customers, CustCity, and Cities tables. Execute the following SQL statement at the mysql command prompt:

```
SELECT CustLN, CityName
FROM Customers AS cu, CustCity AS cc, Cities AS ci
WHERE cu.CustID=cc.CustID AND ci.CityID=cc.CityID;
```

You should receive results similar to the following:

```
+-----------+----------+
| CustLN    | CityName |
+-----------+----------+
| Weatherby | Seattle  |
| Taylor    | Redmond  |
| Cavenaugh | Bellevue |
| Johnson   | Kent     |
| Thomas    | Kent     |
| Delaney   | Kirkland |
+-----------+----------+
6 rows in set (0.00 sec)
```

7. Next, use an `EXPLAIN` statement to analyze the `SELECT` statement that you created in Step 6. Execute the following SQL statement at the mysql command prompt:

```
EXPLAIN SELECT CustLN, CityName
FROM Customers AS cu, CustCity AS cc, Cities AS ci
WHERE cu.CustID=cc.CustID AND ci.CityID=cc.CityID;
```

You should receive results similar to the following:

```
------+--------+---------------+---------+---------+----------------------+------+-
table | type   | possible_keys | key     | key_len | ref                  | rows |
------+--------+---------------+---------+---------+----------------------+------+-
cc    | ALL    | NULL          | NULL    |    NULL | NULL                 |    6 |
cu    | eq_ref | PRIMARY       | PRIMARY |       2 | dvdrentals.cc.CustID |    1 |
ci    | ALL    | PRIMARY       | NULL    |    NULL | NULL                 |    4 |
------+--------+---------------+---------+---------+----------------------+------+-
3 rows in set (0.00 sec)
```

The results shown here represent only a part of the results you will see. Your results should also include the id, select_type, and Extra columns. The id value for each row is 1, the select_type value for each row is `SIMPLE`, and the row for the ci (Cities) table should show an Extra value of `Using where`.

8. Use an `ALTER TABLE` statement to add the necessary indexes to the CustCity table. Execute the following SQL statement at the mysql command prompt:

```
ALTER TABLE CustCity ADD PRIMARY KEY (CustID, CityID),
ADD FOREIGN KEY (CustID) REFERENCES Customers (CustID),
ADD FOREIGN KEY (CityID) REFERENCES Cities (CityID);
```

You should receive a messaging indicating that the statement successfully executed and that six rows were affected.

9. Rerun the EXPLAIN statement that you executed in Step 6. You should receive results similar to the following:

```
------+--------+----------------+---------+---------+----------------------+------+
table | type   | possible_keys  | key     | key_len | ref                  | rows |
------+--------+----------------+---------+---------+----------------------+------+
ci    | ALL    | PRIMARY        | NULL    |    NULL | NULL                 |    5 |
cc    | ref    | PRIMARY,CityID | CityID  |       2 | dvdrentals.ci.CityID |    3 |
cu    | eq_ref | PRIMARY        | PRIMARY |       2 | dvdrentals.cc.CustID |    1 |
------+--------+----------------+---------+---------+----------------------+------+
3 rows in set (0.01 sec)
```

Your results should also include the id, select_type, and Extra columns. The id value for each row is 1, the select_type value for each row is SIMPLE, and the row for the cc (CustCity) table should show an Extra value of Using index.

10. Use the OPTIMIZE TABLE statement to optimize the CustCity table. Execute the following SQL statement at the mysql command prompt:

```
OPTIMIZE TABLE CustCity;
```

You should receive results similar to the following:

```
+---------------------+----------+----------+----------+
| Table               | Op       | Msg_type | Msg_text |
+---------------------+----------+----------+----------+
| dvdrentals.custcity | optimize | status   | OK       |
+---------------------+----------+----------+----------+
1 row in set (0.41 sec)
```

11. Rerun the EXPLAIN statement that you executed in Step 6. You should receive results similar to the following:

```
------+--------+----------------+---------+---------+----------------------+------+
table | type   | possible_keys  | key     | key_len | ref                  | rows |
------+--------+----------------+---------+---------+----------------------+------+
ci    | ALL    | PRIMARY        | NULL    |    NULL | NULL                 |    5 |
cc    | ref    | PRIARY,CityID  | CityID  |       2 | dvdrentals.ci.CityID |    1 |
cu    | eq_ref | PRIMARY        | PRIMARY |       2 | dvdrentals.cc.CustID |    1 |
------+--------+----------------+---------+---------+----------------------+------+
3 rows in set (0.00 sec)
```

Your results should also include the id, select_type, and Extra columns. The id value for each row is 1, the select_type value for each row is SIMPLE, and the row for the cc (CustCity) table should show an Extra value of Using index.

12. Drop the CustCity and Cities tables from the DVDRentals database. Execute the following SQL statements at the mysql command prompt:

```
DROP TABLE CustCity;
DROP TABLE Cities;
```

You should receive messages indicating that the statements were executed successfully.

How It Works

In this exercise, you added the Cities table and the CustCity tables to the DVDRentals database, and then you populated the tables. You then created a `SELECT` statement that retrieved data from the two new tables as well as the Customers table. After you ran the `SELECT` statement, you created the following `EXPLAIN` statement to analyze the `SELECT` statement:

```
EXPLAIN SELECT CustLN, CityName
FROM Customers AS cu, CustCity AS cc, Cities AS ci
WHERE cu.CustID=cc.CustID AND ci.CityID=cc.CityID;
```

The `EXPLAIN` statement is made up of the `EXPLAIN` keyword followed by the `SELECT` statement that you want to analyze. In this case, the `EXPLAIN` statement indicates that 24 rows (6 x 1 x 4) will be examined to process the query. For the CustCity and Cities tables, MySQL will examine all rows in both tables to process the query. For the Customers table, the type is eq_ref, which indicates that MySQL will examine only one row in the Customers table for each combination of rows processed in the other tables. In general, this in itself is an efficient approach, as opposed to examining every row. The fact that every row must be examined in the other two tables is still a problem.

The `EXPLAIN` statement also shows that MySQL will use only the primary key index on the Customers table to process that query, and no indexes from any other table. In an effort to try to improve performance, you used the following `ALTER TABLE` statement to add foreign keys to the CustCity table to reference the Customers and Cities table:

```
ALTER TABLE CustCity ADD PRIMARY KEY (CustID, CityID),
ADD FOREIGN KEY (CustID) REFERENCES Customers (CustID),
ADD FOREIGN KEY (CityID) REFERENCES Cities (CityID);
```

After you modified the tables, you ran the `EXPLAIN` statement once again. This time your results showed that 15 rows would be examined to process the `SELECT` statement, which is an improvement over the 24 rows. In addition, MySQL will use the foreign key index on the CityID column of the CustCity table when processing the query, as well as the primary key index for the Customers table.

Because you altered the CustCity table, you used the following `OPTIMIZE TABLE` statement to ensure that the query optimizer uses the correct cardinality values:

```
OPTIMIZE TABLE CustCity;
```

After you ran this statement, you executed the `EXPLAIN` statement a third time. The results are the same as they were when you executed the statement previously, only this time the results showed that only five rows will be examined (5 x 1 x 1) when processing the query. In reality, at least six rows will be examined. (The `SELECT` statement returns six rows.) As stated earlier, the row values are only estimates of how many rows the optimizer believes will need to be examined. Regardless of the values, the goal is to reduce that number by as many rows as possible.

Optimizing Data Insertion

Because `SELECT` statements are the most common types of SQL statements executed against a MySQL database, optimization efforts tend to focus on improving performance when retrieving data. There might be times, though, when you want to improve the performance of your insert operations, especially

when you want to add many rows of data to your database or must add data often. The following guidelines provide information about several steps that you can take to improve the performance of your insert operations:

- **Use a `LOAD DATA` statement rather than an `INSERT` statement.** Whenever possible, use a `LOAD DATA` statement to insert data from a text file, rather than use an `INSERT` statement. MySQL can add data in a database up to 20 times faster when using a `LOAD DATA` statement, as compared to using an `INSERT` statement. (For information about the `LOAD DATA` statement, see Chapter 11.)
- **Use `INSERT` statements with multiple `VALUES` clauses.** When using `INSERT` statements to add multiple rows in a table, you can use one of two methods to insert that data. The first is to create an `INSERT` statement for each row of data, and the second is to create one `INSERT` statement that contains multiple `VALUES` clauses. Using the second option is much faster because MySQL must process only one SQL statement rather than many statements, and any related indexes must be flushed only once, rather then one time for each `INSERT` statement. (Whenever you execute an `INSERT` statement that affects an indexed column, the related index must be updated and flushed. For information about the `INSERT` statement, see Chapter 6.)
- **When using multiple `INSERT` statements, group them together in a transaction.** There will be times when you must use multiple `INSERT` statements, such as when you're inserting data in multiple tables. In that case, you should isolate your `INSERT` statement in a transaction. This process reduces the number of times that the index must be flushed. (For information about the transactions, see Chapter 12.)
- **Let MySQL insert default values.** When using `INSERT` statements to add data to a table, you can often insert the data without having to specify default values. If this is an option, do not specify those values and instead allow MySQL to insert them. This method results in shorter SQL statements, which means that the server must do less processing on each statement.
- **When possible, use the `DELAYED` option in your `INSERT` statements.** When you specify the `DELAYED` option in an `INSERT` statement, the execution of that statement is delayed until no other client connections are accessing the same table that the `INSERT` statement is accessing. You can continue to take other actions while the `INSERT` statement is in queue. (For information about using the `DELAYED` option in an `INSERT` statement, see Chapter 6.)

When you're dealing with only small amounts of data, most of these steps are inconsequential. However, if you are inserting thousands of rows at a time, performance becomes an important issue. Of all these guidelines, the most important to remember is that bulk loading (by using a `LOAD DATA` statement) is almost always preferable to using `INSERT` statements to add large quantities of data. In addition to improving the performance of your insert operations, you might also find that you need to improve the performance of your data modification operations.

Optimizing Data Modification and Deletion

When considering the steps that you should take to optimize your data modification and deletion operations, take into account the `WHERE` clause in your `UPDATE` and `DELETE` statements. The `WHERE` clause is similar to the `WHERE` clause in the `SELECT` statement in the way in which it determines which rows are examined in each table. As a result, some of the methods that you would use to improve performance of a `SELECT` statement also apply to an `UPDATE` or `DELETE` statement. For example, do not use unnecessary wildcards in your `LIKE` clause, and try to isolate indexed columns used in a comparison expression.

Another consideration with your update and delete operations is how the table is indexed. When you execute an UPDATE or DELETE statement, MySQL optimizes that statement in the same way that it optimizes a SELECT statement, except that there is additional overhead from the actual data modification operations. If columns in your WHERE clause are indexed, those indexes are used to locate the target rows. Because indexes can affect the performance of an update or delete operation, the gains you make by using the index might be lost when modifying the data. Keep in mind, however, that indexes are used primarily to facilitate data retrieval, not data updates and deletions. You should not remove an index if it improves the performance of a few UPDATE or DELETE statements but hurts many SELECT statements. The best way to take indexes into account is to balance the needs of your data retrieval operations against your update and delete operations. In most cases, you want to optimize your system based on your data retrieval operations.

One way to get around this issue of improving data retrieval performance at the expense of data modification operations is to continue to assign indexes based on the performance of your SELECT statement operations, but try to delay updates or deletions until you can perform them at one time, preferably at time when your system's usage is usually low. In addition, enclose your UPDATE statements and DELETE statements in a transaction so that all your data modifications are treated as a unit, which minimizes system processing and index flushing.

One other strategy to consider when deleting data is to use a TRUNCATE statement rather than a DELETE statement when deleting all data from a table. When MySQL processes a TRUNCATE statement, it drops the table and then re-creates it. This process makes the operations extremely fast, much faster than simply trying to remove all the data with a DELETE statement.

For more information about UPDATE, DELETE, *and* TRUNCATE *statements, see Chapter 6.*

In addition to taking steps to improve the performance of your data retrieval and modification operations, you should also take performance into consideration when designing your tables. The choices you make in your table definitions can also affect how well your SQL data modification statements perform.

Optimizing MySQL Tables

When you set up a database, you should take into account your table designs when trying to optimize your system's performance. Of particular importance is how you set up your columns in each table. The following guidelines provide several suggestions for designing your columns:

- **Use identical column types for compared columns.** If you plan to compare columns in a query, as is the case when defining a join condition, use identical data types if possible. For example, if you are defining a foreign key on a table, the foreign key and the referenced column do not have to have identical data types, only compatible data types. This means that a CHAR(6) column can reference a CHAR(8) column. You can then use these two columns to join together the tables. MySQL, however, does not process this join condition as fast as it would if both columns are configured exactly the same, such as making them both CHAR(6). As a result, it is sometimes better to sacrifice a little storage in the interest of improving performance.
- **Specify data types that have the correct length.** When specifying column types, do not specify types with lengths greater than what you need. For example, if you are defining a numerical column, don't use an INT data type if a SMALLINT data type will do. The smaller the column,

the quicker that MySQL can process values used in computations. In addition, the smaller the columns, the smaller the indexes and the more data that can be held in memory.

- **Define your columns as `NOT NULL` when appropriate.** Whenever you can define a column as `NOT NULL`, you should do so. Columns that permit `NULL` values take longer to process than those that do not. If you have a column in which values are often not known, you can still define the column as `NOT NULL`, but you can also define a default value for that column, such as Unknown.
- **Consider defining your columns with the `ENUM` data type.** An `ENUM` data type allows you to specify the values that are permitted in a string column. In some cases, you know exactly what values can be inserted in a column, there are relatively few of those values, and the values will seldom change, if ever. In cases such as this, you should use the `ENUM` data type. Because `ENUM` values are represented internally as numerical values, MySQL can process them much more quickly than a regular string value.

In addition to taking these steps to improve your table designs, you should also use the `OPTIMIZE TABLE` statement to defragment some of your tables once they are created. Earlier in the chapter, you were introduced to the `OPTIMIZE TABLE` statement. One of the functions that the statement performs is to defragment a table. For tables that are modified often or that contain a lot of variable length data (such as is found in `VARCHAR` columns), fragmentation of the table can affect performance. As a result, you should consider running the `OPTIMIZE TABLE` statement against tables of this sort as needed.

As you can see, you can take several steps at the table level to improve the performance of your SQL statements. You can also take steps at the server level to help improve performance, as the following section explains.

Optimizing Your System's Cache

As you learned in Chapter 13, MySQL includes a number of system variables that allow you to specify system variable settings for the MySQL server. Some of the most important of these settings, in terms of the performance of your `SELECT` statements, are those settings related to your query cache. A cache is a place in your system's memory that holds specific types of information. A query cache is a cache that is used specifically to hold the result sets returned by `SELECT` statements.

An application can access information held in memory much faster than information that is stored on a hard disk, particularly if you're accessing large amounts of data stored in a database. As a result, when optimizing performance in MySQL, you should give special consideration to your query cache and the system variables that are used to control that cache.

The query cache speeds up the processing of your `SELECT` statements by caching the result sets retrieved by different queries. When a `SELECT` statement is first executed, that result set is cached. Whenever the same `SELECT` statement is executed, MySQL merely retrieves the data from the cache, without reprocessing the query. If the underlying data is modified in any way, the query results are removed from the cache.

By default, the query cache is not enabled. You can set up query caching on your system by using the following three system variables:

- **query_cache_type:** Specifies the operating mode of the query cache. Three possible values can be assigned to this variable: 0, 1, and 2. A value of 0 (displayed as `OFF`) means do not cache queries. A value of 1 (displayed as `ON`) means cache queries unless the `SQL_NO_CACHE` option is specified in a `SELECT` statement. A value of 2 (displayed as `DEMAND`) means cache queries only if the `SQL_CACHE` option is specified in a `SELECT` statement. By default, this variable is set to 1 (`ON`).
- **query_cache_limit:** Specifies the maximum size that a result set can be in order to be cached. For example, if the limit is set to 2M, no result set larger than 2M will be cached. The default limit is 1M.
- **query_cache_size:** Specifies the amount of memory allocated for caching queries. By default, this variable is set to 0, which means that query caching is turned off. To implement query caching, you should specify a `query_cache_size` setting in the [mysqld] section of your option file. For example, the setting `query_cache_size=10M` enables query caching and allocates 10M of memory to the cache.

As you can see, the only action that you need to take to implement query caching is to set the `query_cache_size` variable, which is set to 0 by default. Of course, you can also change the `query_cache_limit` and `query_cache_type` system variables, but this is not necessary in order to implement query caching. You can set the `query_cache_size` variable at the command line or in an option file, but you cannot use a `SET` statement to specify the cache size. If you do specify a value for the `query_cache_size` variable in an option file, you need to shut down your server and then restart it.

> *The system variables related to your query cache are not the only variables that can affect performance. There are also system variables related to your table and index cache, as well as other components of MySQL. Refer to Chapter 13 and the MySQL product documentation for information about each system variable.*

Now that you have an overview of the system variables related to the query cache, you can enable query caching on your system. The following exercise walks you through the process of viewing the settings for each of these variables and enabling the query cache.

Try It Out Setting Your System's Cache

The following steps describe how to view query cache variables and set the `query_cache_size` system variable:

1. Open the mysql client utility.
2. View the current value set for the `query_cache_type` variable. Execute the following SQL statement at the mysql command prompt:

```
SHOW VARIABLES LIKE 'query_cache_type';
```

You should receive results similar to the following:

```
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | ON    |
+------------------+-------+
1 row in set (0.00 sec)
```

3. View the current value set for the `query_cache_limit` variable. Execute the following SQL statement at the mysql command prompt:

```
SHOW VARIABLES LIKE 'query_cache_limit';
```

You should receive results similar to the following:

```
+------------------+---------+
| Variable_name    | Value   |
+------------------+---------+
| query_cache_limit | 1048576 |
+------------------+---------+
1 row in set (0.00 sec)
```

4. View the current value set for the `query_cache_size` variable. Execute the following SQL statement at the mysql command prompt:

```
SHOW VARIABLES LIKE 'query_cache_size';
```

You should receive results similar to the following:

```
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_size | 0     |
+------------------+-------+
1 row in set (0.00 sec)
```

5. Next, use a text editor such as Vim or Notepad to modify your option file to include a setting for the `query_cache_size` system variable. For Linux users, add a `query_cache_size` entry to the [mysqld] section of the .my.cnf file in the root directory. For Windows users, modify the existing setting in the [mysqld] section of the my.ini file in the `C:\WINDOWS` directory. Your option file should include the following command:

```
query_cache_size=10M
```

After you have modified or added the `query_cache_size` entry, save your option file.

6. Exit the mysql client utility.

7. For the changes in the option file to take effect, you must shut down the MySQL server. In Linux, execute the following command at your operating system's command prompt:

```
mysqladmin -u root -p shutdown
```

When prompted for a password, enter your password and press Enter. The service is stopped.

In Windows, execute the following command at your operating system's command prompt:

```
net stop mysql
```

You should receive a message indicating that the service has been stopped.

8. Now you must restart the MySQL server. In Linux, execute the following command at your operating system's command prompt:

```
mysqld_safe --user=mysql &
```

If, after you execute the `mysql_safe` command, you're not returned to the command prompt right away, you have to press Enter to display the prompt.

In Windows, execute the following command at your operating system's command prompt:

```
net start mysql
```

You should receive a message indicating that the service has been started.

9. Relaunch the mysql client utility.

10. Now view the settings for the `query_cache_size` system variable again. Execute the following SQL statement at the mysql command prompt:

```
SHOW VARIABLES LIKE 'query_cache_size';
```

You should receive results similar to the following:

```
+------------------+----------+
| Variable_name    | Value    |
+------------------+----------+
| query_cache_size | 10485760 |
+------------------+----------+
1 row in set (0.00 sec)
```

11. Close the mysql client utility.

How It Works

All the steps that you took in this exercise should be familiar to you from previous chapters. To begin, you used the `SHOW VARIABLES` statement to view the default settings for each system variable related to the query cache. For example, the following `SHOW VARIABLES` statement retrieved the current setting for the `query_cache_limit` system variable:

```
SHOW VARIABLES LIKE 'query_cache_type';
```

The results indicated that the query cache is on. When you viewed the setting for the `query_cache_limit` variable, you saw that each results set can be 1048576, or 1M. Next you viewed the `query_cache_size` variable, which was set to 0. This meant that no `SELECT` statement result sets were being cached.

Once you verified the settings for all three system variables, you added or updated the following setting in the [mysqld] section of your option file:

```
query_cache_size=10M
```

This entry sets the query cache size to 10M. You implemented the new setting by stopping the server and then restarting it. From there, you opened the MySQL client utility and used the `SHOW VARIABLES` statement once more to verify the `query_cache_size` system setting. The results indicated that the new setting had been implemented. Now your `SELECT` statements will be cached, which should improve the performance of your `SELECT` statements. You could have also specified different settings for the `query_cache_type` and `query_cache_limit` system variables in your option file, but it wasn't necessary. Because you didn't, the default values for both variables are used.

Summary

As you have learned in this chapter, there are many steps that you can take to try to improve how well your SQL statements perform. The most important step is to ensure that your tables have been properly indexed. You want to be sure that each table contains the indexes that it needs, while at the same time ensuring that the table is not over-indexed. When necessary, you should also analyze your `SELECT` statements to ensure that they are being executed as optimally as possible. You should also take the steps necessary to maximize the performance of your data modification statements. In addition, you should look at how your tables have been created, and you should consider enabling query caching. To help you optimize your system, this chapter covered the following topics:

- Setting up effective indexes
- Using the `EXPLAIN` statement to analyze `SELECT` statement performance
- Using the `OPTIMIZE TABLE` statement to optimize your table after changes have been made to that table
- Improving the performance of data retrieval and modification
- Enabling your system's query cache to improve query performance

The subject of optimization is a broad one that can cover many aspects of running MySQL. Although this chapter attempted to touch on many of the important issues concerning optimizing your system, it could not cover each subject as extensively as possible. In fact, the subject of optimization is a book in itself. In addition, this chapter does not cover hardware considerations in system optimization. Consequently, you're encouraged to refer to other resources for information about optimizing your system, particularly the MySQL product documentation and Web site (`www.mysql.com`). From this chapter, you should have been able to gain a solid foundation in understanding many of the steps that you can take to optimize your SQL statements. From here, you're ready to move on to the topics of replicating, backing up, and restoring your MySQL database, which are covered in Chapter 16.

Exercises

This chapter explains several of the steps that you can take to optimize your SQL statements. The following exercises help you build on your knowledge of optimization. To view the solutions to these exercises, see Appendix A.

1. You plan to execute the following `SELECT` statement for an application that you're developing:

```
SELECT PartID, PartName, ManfName
FROM Parts AS p, Manufacturers as m
WHERE p.ManfID = m.ManfID
ORDER BY PartName;
```

   The PartID column of the Parts table is configured as the primary key, and the ManfID column of the Manufacturers table is configured as the primary key. You anticipate that the `SELECT` statement will be executed frequently. On which columns should you consider creating indexes?

2. You want to analyze the `SELECT` statement shown in Exercise 1 to determine how MySQL will process the statement. What statement should you use to analyze the `SELECT` statement?

3. You use an `EXPLAIN` statement to analyze a `SELECT` statement. The analysis shows that the statement is not using one of the indexes defined on the table. What can you do to force the `SELECT` statement to use that index?

4. You must insert a large amount of data in one of the tables in your database. What is the fastest way to insert that data?

5. You plan to delete all the data from the Parts table. You want the deletion to be executed as quickly as possible, and you don't need to know how many rows have been deleted. What SQL statement should you use to delete the data?

6. You are planning the table structure for a MySQL database. You want to ensure that your columns are defined to ensure the maximum performance. What guidelines should you follow when setting up your columns?

7. You want to implement query caching, and you want to ensure that the cache grows no larger than 10M. What should you do to implement query caching?