11

Exporting, Copying, and Importing Data

Up to this point in the book, the process of managing data has been confined to the manipulation of data in your database. For example, to add data to your tables, you manually created INSERT statements that targeted specific tables. To view data, you manually created SELECT statements that retrieved data from specific tables. In each case, the data was added to the tables by specifying those values to be inserted, or the data was retrieved by executing the applicable SELECT statement each time you wanted to view that data. At no time was data copied to or from files outside the database, nor was data copied between tables in the database.

The limitations of these approaches become apparent when you want to add large quantities of data to a database or manage large quantities of data outside the database. MySQL supports a number of SQL statements and commands that allow you to export data into files outside the database, copy data between tables in a database, and import data into the database. By using these statements and commands, you can easily work with large amounts of data that must be added to and retrieved from a database or data that must be copied from one table to the next. This chapter discusses how to use these statements and commands and provides examples of each. Specifically, the chapter explains how to perform the following tasks:

- ❑ Create SELECT statements that export data to out files and dump files
- ❑ Create INSERT and REPLACE statements that copy data into existing tables and create CREATE TABLE statements that copy data into new tables
- ❑ Create LOAD DATA statements and execute mysql and mysqlimport commands that import data into existing tables

Exporting Data Out of a Table

There might be times when you want to export data in a MySQL database to text files stored outside the database. MySQL allows you to export this data by using SELECT statements that include the appropriate export definitions. To understand how to add an export definition to a SELECT statement, return to the SELECT statement syntax that you were introduced to in Chapter 7:

TEAM LinG - Live, Informative, Non-cost and Genuine !

```
<select statement>::=
SELECT
[<select option> [<select option>...]]
{* | <select list>}
[<export definition>]
[
   FROM <table reference> [{, <table reference>}...]
   [WHERE <expression> [{<operator> <expression>}...]]
   [GROUP BY <group by definition>]
   [HAVING <expression> [{<operator> <expression>}...]]
   [ORDER BY <order by definition>]
   [LIMIT [<offset>,] <row count>]
   [PROCEDURE <procedure name> [(<argument> [{, <argument>}...])]]
   [{FOR UPDATE} | {LOCK IN SHARE MODE}]
]

<export definition>::=
INTO OUTFILE '<filename>' [<export option> [<export option>]]
| INTO DUMPFILE '<filename>'

<export option>::=
{FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']}
| {LINES
   [STARTING BY '<value>']
   [TERMINATED BY '<value>']}
```

The syntax shown here includes only the basic elements that make up a `SELECT` statement and those elements specific to exporting data. (For a detailed explanation of the `SELECT` statement, refer to Chapter 7.) Notice that the *`<export definition>`* placeholder precedes the `FROM` clause. To export data, you must include an export definition that specifies whether the data is imported to an out file or a dump file. An *out file* is a text file (such as a .txt or .sql file) that contains one or more rows of exported data in a delimited format. A *delimited* format is one in which the values and rows are separated and enclosed by specific types of characters. For example, a tab is commonly used to separate values in the same row, and a line break (also known as newline) is often used to separate rows. In contrast to an out file, a *dump file* is a text file that contains only one row that is not delimited. Dump files are used primarily for large values, such as a value from a column configured with a `BLOB` data type.

In this section, you learn about how to export data to an out file and to a dump file. To demonstrate how to export data, the table shown in the following table definition is used:

```
CREATE TABLE CDs
(
   CDID SMALLINT NOT NULL PRIMARY KEY,
   CDName VARCHAR(50) NOT NULL,
   InStock SMALLINT UNSIGNED NOT NULL,
   Category VARCHAR(20)
);
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

In addition to using the CDs table to demonstrate how to export data, the table is also used later in the chapter for examples that show how to copy and import. For all examples in this chapter, you can assume that the following `INSERT` statement has been used to populate the CDs table:

```
INSERT INTO CDs
VALUES (101, 'Bloodshot', 10, 'Rock'),
(102, 'New Orleans Jazz', 17, 'Jazz'),
(103, 'Music for Ballet Class', 9, 'Classical'),
(104, 'Music for Solo Violin', 24, NULL),
(105, 'Mississippi Blues', 2, 'Blues'),
(106, 'Mud on the Tires', 12, 'Country'),
(107, 'The Essence', 5, 'New Age'),
(108, 'The Magic of Satie', 42, 'Classical'),
(109, 'La Boheme', 20, 'Opera'),
(110, 'Ain\'t Ever Satisfied', 23, 'Country'),
(111, 'Live in Paris', 18, 'Jazz'),
(112, 'Richland Woman Blues', 22, 'Blues'),
(113, 'Stages', 42, 'Blues');
```

Now take a look how to export data to out files and dump files.

Exporting Data to an Out File

As you saw in the previous section, the elements in the `SELECT` statement syntax that are specific to exporting to an out file are as follows:

```
INTO OUTFILE '<filename>' [<export option> [<export option>]]

<export option>::=
{FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']}
| {LINES
   [STARTING BY '<value>']
   [TERMINATED BY '<value>']}
```

When exporting data to an out file, you must specify the `INTO OUTFILE` clause after the select list and before the `FROM` clause. The `INTO OUTFILE` clause includes a filename, enclosed in single quotes, and one or two export options. The export options can include a `FIELDS` clause, a `LINES` clause, or both. If you include both, then the `FIELDS` clause must come before the `LINES` clause. In addition, for each clause that you specify, you must specify at least one of the subclauses available for that clause. For example, if you specify a `LINES` clause, you must also specify a `STARTING BY` subclause, a `TERMINATED BY` subclause, or both.

Now take a close look at the `FIELDS` clause, which includes three optional subclauses:

```
FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']
```

As you can see in the syntax, the FIELDS clause includes the following three subclauses:

- **TERMINATED BY:** Specifies the character or characters to be used to separate each value returned by a row of data. By default, a tab is used, which is indicated by \t.
- **ENCLOSED BY:** Specifies the character to be used to enclose each value returned by a row of data. By default, no characters are used.
- **ESCAPED BY:** Specifies the character to be used to escape special characters. By default, a backslash is used, which is indicated by \\.

In general, you would use these subclauses only when you have a special requirement for the way data is retrieved from the file. Otherwise, the default values are normally adequate. One thing to note, however, is that the TERMINATED BY clause allows you to specify more than one character to use to separate values in a row. The other subclauses allow you to specify only one character.

For the most part, the TERMINATED BY and ENCLOSED BY clauses are fairly straightforward. Whatever values you specify in these clauses are the values that are used. It's worth taking a closer look at the ESCAPED BY subclause. This clause is used to escape any characters in the data that are also used in the FIELDS subclauses. In other words, the characters in the data values are treated literally, rather than the way they are treated between values or rows. The ESCAPED BY subclause escapes the following characters when they appear in a value:

- The first character in the FIELDS TERMININATED BY and LINES TERMINATED BY subclauses
- The character in the ENCLOSED BY subclause
- The character in the ESCAPED BY subclause
- Uppercase N, when used to represent NULL

Whenever any of these characters appear in a data value, the character specified in the ESCAPED BY subclause precedes the character. As a result, when you import data from the out file, you have a way to ensure that specified characters are treated appropriately, depending on where they appear in the text file. Later in this section, you see an example of how this works.

Now take a look at the LINES clause, shown in the following syntax:

```
LINES
   [STARTING BY '<value>']
   [TERMINATED BY '<value>']
```

As you can see, the LINES clause supports the following two subclauses:

- **STARTED BY:** Specifies the character or characters to be used to begin each row of data. By default, no characters are used.
- **TERMINATED BY:** Specifies the character or characters to be used to end each row of data. By default, a newline is used, which is shown as \n.

When you're specifying the values that can be inserted in any of the FIELDS or LINES subclauses, you can use a literal value, or you can use one of the special values supported by the subclauses. Whenever you use a special value, it must be preceded by a backslash. The following table provides the meanings for the primary special values that you can use in your FIELDS and LINES subclauses.

TEAM LinG - Live, Informative, Non-cost and Genuine !

Value	Meaning
`\'`	Single quote
`\\`	Backslash
`\n`	Newline
`\r`	Carriage return
`\s`	Space
`\t`	Tab

Whenever you specify one of the special values, that character is used for the purpose specified by the subclause. For example, if you specify that a space (`\s`) be used to separate values (the `FIELDS TERMINATED BY` subclause), a space is used between each value in each row.

Now that you have an overview of the components that make up an export definition, you can examine an example of a `SELECT` statement that includes a basic definition. In the following `SELECT` statement, values are retrieved from the CDs table and exported to a file named CDsOut.txt:

```
SELECT CDName, InStock, Category
   INTO OUTFILE 'CDsOut.txt'
FROM CDs;
```

As you can see, the statement includes an `INTO OUTFILE` clause directly after the select list and preceding the `FROM` clause. Because the `INTO OUTFILE` clause is used, a file is created and the values are added to that file, rather than a result set being generated. Any values that you would expect to see in the result set are copied to the file. If a file by that name already exists, the `SELECT` statement fails.

The file is saved to the location of the database folder that corresponds to the database in which you're working. For example, if you're working in the test database, the file is saved to the test folder in your data directory. To save the file to a different location, you must specify a path for that file.

Once an out file has been created, you can open the file and view its contents. For example, if you were to view the contents of the CDsOut.txt file, you would see results similar to the following:

```
Bloodshot       10      Rock
New Orleans Jazz17      Jazz
Music for Ballet Class  9       Classical
Music for Solo Violin   24      \N
Mississippi Blues       2       Blues
Mud on the Tires12      Country
The Essence     5       New Age
The Magic of Satie      42      Classical
La Boheme       20      Opera
Ain't Ever Satisfied    23      Country
Live in Paris   18      Jazz
Richland Woman Blues    22      Blues
Stages   42     Blues
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

As you can see, the results that you would expect to have been generated by the result set are added to the file. Note that tabs separate the values and that each row is on its own line. In addition, the values are not enclosed in any specific character. These are the results you would expect when you generate an out file without specifying any `FIELDS` or `LINES` subclauses.

In some text programs, the rows might not be shown as being separated into different lines, but rather separated by a symbol that looks like a small rectangle. As a result, the rows run together in one line that wraps down your document. If you were to copy and paste the results into a word processing program, you would find that they are displayed with each row on a separate line, as they are in the preceding example.

One thing that you might notice is that, in the fourth row, the Category value is `\N`. This indicates that the query returned a `NULL`. By default, whenever a value is `NULL`, an uppercase N is returned, preceded by a backslash. If the `ESCAPED BY` option specifies a character other than a backslash, that character precedes the N.

The next example `SELECT` statement is similar to the last except that two `FIELDS` subclauses are specified, as shown in the following statement:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsOut.txt'
   FIELDS
      TERMINATED BY ','
      ENCLOSED BY '"'
FROM CDs;
```

In this case, the values are saved to a file named CDsOut.txt, the values are separated by commas (as specified in the `TERMINATED BY` subclause), and the values are enclosed in double quotes (as specified in the `ENCLOSED BY` subclause).

If you try to create an out file and that file already exists, you receive an error. If the file already exists, you should rename the original out file or delete it before attempting to create the new file.

The following results show the contents of the CDsOut.txt file:

```
"Bloodshot","10","Rock"
"New Orleans Jazz","17","Jazz"
"Music for Ballet Class","9","Classical"
"Music for Solo Violin","24",\N
"Mississippi Blues","2","Blues"
"Mud on the Tires","12","Country"
"The Essence","5","New Age"
"The Magic of Satie","42","Classical"
"La Boheme","20","Opera"
"Ain't Ever Satisfied","23","Country"
"Live in Paris","18","Jazz"
"Richland Woman Blues","22","Blues"
"Stages","42","Blues"
```

As you can see, commas have replaced the tabs and each value is enclosed in double quotes. Now modify this statement even further by adding an `ESCAPED BY` subclause:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsOut.txt'
   FIELDS
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

```
        TERMINATED BY ','
        ENCLOSED BY '\"'
        ESCAPED BY '\''
    FROM CDs;
```

The ESCAPED BY value specifies that a single quote be used to escape the subclause values. As a result, each comma, double quote, and single quote that appears in a value is preceded by a single quote. The following results show the contents of the CDsOut.txt file:

```
"Bloodshot","10","Rock"
"New Orleans Jazz","17","Jazz"
"Music for Ballet Class","9","Classical"
"Music for Solo Violin","24",'N
"Mississippi Blues","2","Blues"
"Mud on the Tires","12","Country"
"The Essence","5","New Age"
"The Magic of Satie","42","Classical"
"La Boheme","20","Opera"
"Ain''t Ever Satisfied","23","Country"
"Live in Paris","18","Jazz"
"Richland Woman Blues","22","Blues"
"Stages","42","Blues"
```

First, notice that the uppercase N is now preceded by a single quote, rather than a backslash. In addition, the single quote (used as an apostrophe) in the tenth row is also preceded by a single quote because a single quote is one of the values specified in the FIELDS subclauses. As a result, the second single quote is treated as a literal value when importing data from this file.

Now that you've seen several examples of how to use FIELDS subclauses, you can take a look at an example of LINES subclauses. The following statement is similar to the last, only now it includes LINES subclauses rather than FIELD subclauses:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsOut.txt'
   LINES
      STARTING BY '*'
      TERMINATED BY '**'
FROM CDs
WHERE Category='Blues' OR Category='Jazz';
```

The export definition in this statement now specifies that each row should begin with an asterisk (*) and each row should end with double asterisks (**). The following results show the contents of the CDsOut.txt file:

```
*New Orleans Jazz       17      Jazz***Mississippi Blues       2       Blues***Live
in Paris 18      Jazz***Richland Woman Blues    22      Blues***Stages  42
Blues**
```

As you can see, each row is no longer on its own line, and the rows appear to be separated by three asterisks, making it difficult to distinguish where one row ends and where the other begins. To put the rows on separate lines, you must modify the TERMINATED BY subclause, as shown in the following SELECT statement:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsOut.txt'
   LINES
      STARTING BY '*'
      TERMINATED BY '**\n'
FROM CDs
WHERE Category='Blues' OR Category='Jazz';
```

Notice that the `TERMINATED BY` clause now includes the newline (`\n`) value, as well as the double asterisks. As a result, the contents of the CDsOut.txt file now appear similar to the following:

```
*New Orleans Jazz       17      Jazz**
*Mississippi Blues      2       Blues**
*Live in Paris  18      Jazz**
*Richland Woman Blues   22      Blues**
*Stages  42      Blues**
```

As you can see, each row is on a separate line, begins with an asterisk, and ends with double asterisks. You're not limited to only symbols in your `TERMINATED BY` subclauses. The following `SELECT` statement terminates each line with `<<end>>`:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsOut.txt'
   FIELDS
      TERMINATED BY ','
      ENCLOSED BY '\"'
   LINES
      TERMINATED BY '<<end>>\n'
FROM CDs
WHERE Category='Blues' OR Category='Jazz';
```

Now when you view the contents of the CDsOut.txt file, you see the following results:

```
"New Orleans Jazz","17","Jazz"<<end>>
"Mississippi Blues","2","Blues"<<end>>
"Live in Paris","18","Jazz"<<end>>
"Richland Woman Blues","22","Blues"<<end>>
"Stages","42","Blues"<<end>>
```

As you can see, you can be quite imaginative with the values in your `FIELDS` and `LINES` subclauses; however, you should use these clauses only if it's necessary to set up the file in a specific way. Most of the time, the default settings are adequate.

In the following exercise, you create several `SELECT` statements that each include an export definition that creates an out file. Because you are exporting data from the DVDRentals database, the files are saved to the DVDRentals folder in the data directory.

Try It Out Exporting DVDRentals Data to an Out File

The following steps describe how to create the `SELECT` statements:

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

```
use DVDRentals
```

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

TEAM LinG - Live, Informative, Non-cost and Genuine !

2. The first `SELECT` statement exports data from joined tables in the DVDRentals database to a text file named AvailDVDs.txt. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO OUTFILE 'AvailDVDs.txt'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s2'
ORDER BY DVDName;
```

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

3. Go to the DVDRentals folder of your data directory, and open the AvailDVDs.txt file. The file should contain the following values in a format similar to what is shown here:

```
Amadeus  Drama   Widescreen     PG
Mash     Comedy  Widescreen     R
The Maltese Falcon      Drama   Widescreen     NR
The Rocky Horror Picture Show   Comedy  Widescreen     NR
What's Up, Doc? Comedy  Widescreen     G
```

4. In the next `SELECT` statement, you add the `FIELDS TERMINATED BY` and `LINES TERMINATED BY` subclauses. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO OUTFILE 'AvailDVDs2.txt'
   FIELDS TERMINATED BY '*,*'
   LINES TERMINATED BY '**\n'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s2'
ORDER BY DVDName;
```

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

5. Go to the DVDRentals folder of your data directory and open the AvailDVDs2.txt file. The file should contain the following values in a format similar to what is shown here:

```
Amadeus*,*Drama*,*Widescreen*,*PG**
Mash*,*Comedy*,*Widescreen*,*R**
The Maltese Falcon*,*Drama*,*Widescreen*,*NR**
The Rocky Horror Picture Show*,*Comedy*,*Widescreen*,*NR**
What's Up, Doc?*,*Comedy*,*Widescreen*,*G**
```

How It Works

The first `SELECT` statement that you created includes a basic export definition:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO OUTFILE 'AvailDVDs.txt'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s2'
ORDER BY DVDName;
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

The export definition is made up of the `INTO OUTFILE` keywords and the name of the new file, enclosed in single quotes. Because you specified no `FIELDS` or `LINES` subclauses, the values are added using the default format. As a result, values are separated with tabs, the values are not enclosed in any characters, and special characters are escaped by the use of a backslash. In addition, each row is on its own line, and no characters precede each row.

The next statement that you created is similar to the last, except that it includes the `FIELDS TERMINATED BY` and `LINES TERMINATED BY` subclauses:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO OUTFILE 'AvailDVDs2.txt'
   FIELDS TERMINATED BY '*,*'
   LINES TERMINATED BY '**\n'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s2'
ORDER BY DVDName;
```

Because you added the subclauses to the export definition, the fields are now separated by an asterisk, a comma, and another asterisk, and the rows end in double asterisks, although they are still each on their own lines.

Exporting Data to a Dump File

Exporting data to a dump file is much simpler than an out file because you cannot specify any `FIELDS` or `LINES` subclauses. As a result, the syntax for a dump file export definition is very simple, as shown in the following:

```
INTO DUMPFILE '<filename>'
```

As you can see, you need to specify only the `INTO DUMPFILE` keywords and the filename, enclosed in single quotes. As with an out file, the dump file export definition is placed after the `SELECT` list and before the `FROM` clause of the `SELECT` statement, as shown in the following example:

```
SELECT CDName, InStock, Category
   INTO DUMPFILE 'CDsOut.txt'
FROM CDs
WHERE CDID=110;
```

In this statement, the data extracted from the CDs table is exported to the CDsOut.txt file. Notice that the `SELECT` statement returns only one row of data. Your `SELECT` statement must return exactly one row if you want to save data to a dump file. In addition, the values are not in any way delimited. For example, the preceding statement returns the row for the CD named *Ain't Ever Satisfied*. When added to the file, all values from that row are run together, as shown in the following results:

```
Ain't Ever Satisfied23Country
```

As you can see, the only spaces are those that are between the words in a value, but there is nothing that separates the individual values from one another. Normally, you would create a dump file if you want to store a value from a `BLOB` column in a file.

TEAM LinG - Live, Informative, Non-cost and Genuine !

In this Try It Out exercise, you create a SELECT statement that includes an export definition. The statement extracts data from joined tables in the DVDRentals database and adds that data to a dump file.

Try It Out Exporting DVDRentals Data to a Dump File

The following steps describe how to create the 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 switched to the DVDRentals database.

2. In this statement, you extract data about the DVD *Out of Africa* and place it in a dump file. Execute the following SQL statement at the mysql command prompt:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO DUMPFILE 'DVD3.txt'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND DVDID=3;
```

   You should receive a message indicating that the statement executed successfully, affecting one row.

3. Go to the DVDRentals folder of your data directory, and open the DVD3.txt file. The file should contain the following values in a format similar to what is shown here:

```
Out of AfricaDramaWidescreenPG
```

How It Works

The SELECT statement that you included in this exercise includes an export definition that exports data to a dump file:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO DUMPFILE 'DVD3.txt'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND DVDID=3;
```

In this statement, one row of data is retrieved from joined tables in the DVDRentals database. When the statement was executed, a dump file named DVD3.txt was created and the data retrieved by the SELECT statement was inserted in the dump file. The values added to the file were not delimited in any way, so the values ran together. The only spaces anywhere in the text were those that are part of the *Out of Africa* value retrieved from the DVDName column.

Copying Data into a Table

MySQL provides two primary methods for copying existing data into a table. You can add data to a new table when the table is created, or you can add data to an existing table. In this section, you learn how to use a CREATE TABLE statement to add data to a new table and how to use an INSERT statement to add data to an existing table.

TEAM LinG - Live, Informative, Non-cost and Genuine !

Copying Data into a New Table

When you learned about how to create tables in Chapter 5, you were introduced to the syntax used for table definitions. The basic syntax for a table definition is as follows:

```
<table definition>::=
CREATE TABLE <table name>
(<table element> [{, <table element>}...])
[<table option> [<table option>...]]
[<select statement>]
```

The syntax here includes an element that you did not see in Chapter 5, the *`<select statement>`* placeholder. (For an explanation of the other components of a `CREATE TABLE` statement, see Chapter 5.) As the placeholder indicates, you can add a `SELECT` statement to the end of your table definition. The values returned by that statement are automatically inserted in the new table. To add a `SELECT` statement, the statement must return results that can be inserted naturally in the statement. This means that the values must be compatible with the data types of the columns in the new table, and each row must contain the correct number of values, one for each column of the new table.

For example, the following `CREATE TABLE` statement extracts data from the CDs table and adds it to the new CDs2 table:

```
CREATE TABLE CDs2
(
   CDID SMALLINT NOT NULL PRIMARY KEY,
   CDName VARCHAR(50) NOT NULL,
   InStock SMALLINT UNSIGNED NOT NULL
)
SELECT CDID, CDName, InStock
FROM CDs
WHERE Category='Blues' OR Category='Jazz';
```

The first part of the statement is a standard table definition. The table named CDs2 includes three columns: CDID, CDName, and InStock. The next part of the table definition (the last three rows) includes the `SELECT` statement that retrieves the values to be inserted in the new table.

Once the table has been created, you can run the following `SELECT` statement:

```
SELECT * FROM CDs2;
```

Your statement should return results similar to the following:

```
+------+----------------------+---------+
| CDID | CDName               | InStock |
+------+----------------------+---------+
|  102 | New Orleans Jazz     |      17 |
|  105 | Mississippi Blues    |       2 |
|  111 | Live in Paris        |      18 |
|  112 | Richland Woman Blues |      22 |
|  113 | Stages               |      42 |
+------+----------------------+---------+
5 rows in set (0.00 sec)
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

As you can see, these are the same results that you would have received if you executed the `SELECT` statement separately from the `CREATE TABLE` statement. By adding the `SELECT` statement to the `CREATE TABLE` statement, you save yourself the trouble of first creating the table and then using an `INSERT` statement to add data to that table.

Keep in mind that when you create a table, the columns must be defined with data types compatible with the data that you're retrieving from an existing table. This refers not only to the type of data (for example, `INT` versus `CHAR`), but also to the length of the data (for example, `CHAR(4)` versus `CHAR(40)`). If you use an incorrect data type or specify the size too small for the data type, MySQL truncates the data. This means that the data is shortened or another value is inserted in the column. For example, suppose that you use the following `CREATE TABLE` statement to create a table:

```
CREATE TABLE CDs2a
(
   CDID SMALLINT NOT NULL PRIMARY KEY,
   CDName VARCHAR(5) NOT NULL,
   InStock SMALLINT UNSIGNED NOT NULL
)
SELECT CDID, CDName, InStock
FROM CDs
WHERE Category='Blues' OR Category='Jazz';
```

Notice that the CDName column of the new table is defined with a `VARCHAR(5)` data type. The data that is retrieved for that column comes from a column with a `VARCHAR(50)` data type, and values in that column all exceed five characters. As a result, when you try to insert data it is truncated. For example, if you retrieve all the data from the CDs3 table, you receive results similar to the following:

```
+------+--------+---------+
| CDID | CDName | InStock |
+------+--------+---------+
|  102 | New O  |      17 |
|  105 | Missi  |       2 |
|  111 | Live   |      18 |
|  112 | Richl  |      22 |
|  113 | Stage  |      42 |
+------+--------+---------+
5 rows in set (0.00 sec)
```

As you can see, each CDName value has been truncated, which means that only the first five characters of the name are displayed.

In this Try It Out, you create a table named DVDs2 in the DVDRentals database. The table is populated with data returned by a `SELECT` statement that joins several tables in the database.

Try It Out Copying Data to a New Table in the DVDRentals Database

Follow these steps to create the table definition:

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

```
use DVDRentals
```

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

2. Now create a table definition that includes a SELECT statement that retrieves data from the DVDs, MovieTypes, and Formats tables. Execute the following SQL statement at the mysql command prompt:

```
CREATE TABLE DVDs2
(
   DVDName VARCHAR(60) NOT NULL,
   MTypeDescrip VARCHAR(30) NOT NULL,
   FormDescrip VARCHAR(15) NOT NULL,
   RatingID VARCHAR(4) NOT NULL
)
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s2'
ORDER BY DVDName;
```

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

3. Next, view the contents of the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
SELECT * FROM DVDs2;
```

You should receive results similar to the following:

```
+-------------------------------+--------------+-------------+----------+
| DVDName                       | MTypeDescrip | FormDescrip | RatingID |
+-------------------------------+--------------+-------------+----------+
| Amadeus                       | Drama        | Widescreen  | PG       |
| Mash                          | Comedy       | Widescreen  | R        |
| The Maltese Falcon            | Drama        | Widescreen  | NR       |
| The Rocky Horror Picture Show | Comedy       | Widescreen  | NR       |
| What's Up, Doc?               | Comedy       | Widescreen  | G        |
+-------------------------------+--------------+-------------+----------+
5 rows in set (0.01 sec)
```

Be sure not to delete the DVDs2 table from the DVDRentals database because you use it for Try It Out sections later in the chapter.

How It Works

In this exercise, you created a table definition that includes a SELECT statement at the end of the definition:

```
CREATE TABLE DVDs2
(
   DVDName VARCHAR(60) NOT NULL,
   MTypeDescrip VARCHAR(30) NOT NULL,
   FormDescrip VARCHAR(15) NOT NULL,
   RatingID VARCHAR(4) NOT NULL
)
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
FROM DVDs AS d, MovieTypes AS m, Formats AS f
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

```
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s2'
ORDER BY DVDName;
```

The statement first creates the actual table, which is named DVDs2 and which contains four columns: DVDName, MTypeDescrip, FormDescrip, and RatingID. After the table is defined, the table definition then includes a `SELECT` statement that retrieves data from the DVDs, MovieTypes, and Formats tables. The data retrieved by the `SELECT` statement is inserted in the DVDs2 table. Notice that each row returned by the `SELECT` statement contains the same number of values as there are columns in the new table. In addition, the data returned by the `SELECT` statement is made up of values that are consistent with the column types of the new table.

Copying Data into an Existing Table

MySQL provides two methods that you can use to copy data into an existing table: the `INSERT` statement and the `REPLACE` statement. (You learned about both these statements in Chapter 6.) Both statements are used to add data to a table. The main difference between the two is in how values in a primary key column or a unique index are treated. In an `INSERT` statement, if you try to insert a row that contains a unique index or primary key value that already exists in the table, you won't be able to add that row. With a `REPLACE` statement, the old row is deleted and the new row is added. In all other respects, the statements are the same.

Using the INSERT Statement to Copy Data

When Chapter 6 introduced you to the `INSERT` statement, it provided you with the statement syntax. The following syntax provides you with an abbreviated version of what you saw in Chapter 6:

```
<insert statement>::=
INSERT [LOW_PRIORITY | DELAYED] [IGNORE] [INTO]
{<values option> | <set option> | <select option>}

<select option>::=
<table name> [(<column name> [{, <column name>}...])]
<select statement>
```

As you can see, the main part of the statement includes three options: the `<values option>`, the `<set option>`, and the `<select option>`. Of these three, the `<select option>` is the only one shown here. For an explanation of the other two options, see Chapter 6.

The `<select option>` allows you to define a `SELECT` statement in your `INSERT` statement. The values returned by the `SELECT` statement are then inserted in the table specified by the main part of the `INSERT` statement. For example, the following `INSERT` statement adds data from the CDs table to the CDs2 table:

```
INSERT INTO CDs2
SELECT CDID, CDName, InStock FROM CDs
WHERE Category='Country' OR Category='Rock';
```

In this statement, the `INSERT` clause includes the name of the target table (CDs2). (This is the table that you created when learning about adding data to a new table.) As you can see, the `SELECT` statement follows the `INSERT` clause. In this case, the statement retrieves data from the CDID, CDName, and InStock columns of the CDs table and returns three rows (with CDID values of 101, 106, and 110). The SELECT statement must return the same number of columns specified in the INSERT clause. If no columns are specified in the INSERT clause, the SELECT statement must return the same number of columns that exist in the target table. In addition, the values retrieved by the SELECT statement must be compatible with the data types of the targeted columns.

Once you execute the INSERT statement, you can use the following SELECT statement to view the contents of the CDs2 table:

```
SELECT * FROM CDs2;
```

The SELECT statement returns the following result set:

```
+------+---------------------+---------+
| CDID | CDName              | InStock |
+------+---------------------+---------+
|  102 | New Orleans Jazz    |      17 |
|  105 | Mississippi Blues   |       2 |
|  111 | Live in Paris       |      18 |
|  112 | Richland Woman Blues |      22 |
|  113 | Stages              |      42 |
|  101 | Bloodshot           |      10 |
|  106 | Mud on the Tires    |      12 |
|  110 | Ain't Ever Satisfied |      23 |
+------+---------------------+---------+
8 rows in set (0.00 sec)
```

As you can see, the CDs2 table now contains the rows added when the table was created and the rows returned by the SELECT statement specified in the INSERT statement.

Using the REPLACE Statement to Copy Data

The REPLACE statement syntax is essentially the same as that of the INSERT statement, as shown in the following:

```
<replace statement>::=
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
{<values option> | <set option> | <select option>}

<select option>::=
<table name> [(<column name> [{, <column name>}...])]
<select statement>
```

To use a REPLACE statement to insert values returned by a SELECT statement, you should add the SELECT statement after the REPLACE clause, in the same way as you did with the INSERT statement. For example, the following REPLACE statement is identical to the INSERT statement in the previous example, except that it now uses the keyword REPLACE:

```
REPLACE INTO CDs2
SELECT CDID, CDName, InStock
FROM CDs
WHERE Category='Country' OR Category='Rock';
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

The `SELECT` statement returns the same results that it returned in the `INSERT` statement, and those values are inserted in the CDs2 table.

In this Try It Out section, you create an `INSERT` statement that adds data to the DVDs2 table that you created in the last Try It Out section. The `INSERT` statement uses data retrieved from several joined tables in the DVDRentals database.

Try It Out Copying Data to an Existing Table in the DVDRentals Database

The following steps describe how to create the `INSERT` statement:

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

```
use DVDRentals
```

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

2. Now create the `INSERT` statement, which includes the necessary `SELECT` statement. Execute the following SQL statement at the mysql command prompt:

```
INSERT INTO DVDs2
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s1'
ORDER BY DVDName;
```

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

3. Now view the contents of the DVDs2 table to verify that the new data has been added. Execute the following SQL statement at the mysql command prompt:

```
SELECT * FROM DVDs2;
```

   You should receive results similar to the following:

```
+-------------------------------+--------------+-------------+----------+
| DVDName                       | MTypeDescrip | FormDescrip | RatingID |
+-------------------------------+--------------+-------------+----------+
| Amadeus                       | Drama        | Widescreen  | PG       |
| Mash                          | Comedy       | Widescreen  | R        |
| The Maltese Falcon            | Drama        | Widescreen  | NR       |
| The Rocky Horror Picture Show | Comedy       | Widescreen  | NR       |
| What's Up, Doc?               | Comedy       | Widescreen  | G        |
| A Room with a View            | Drama        | Widescreen  | NR       |
| Out of Africa                 | Drama        | Widescreen  | PG       |
| White Christmas               | Musical      | Widescreen  | NR       |
+-------------------------------+--------------+-------------+----------+
8 rows in set (0.00 sec)
```

How It Works

In this exercise, you created an `INSERT` statement that included an `INSERT` clause and a `SELECT` statement:

```
INSERT INTO DVDs2
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
```

```
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND StatID='s1'
ORDER BY DVDName;
```

The `INSERT` clause includes only the `INSERT INTO` keywords and the name of the target table, DVDs2. The rest of the statement is a `SELECT` statement that retrieves data from the DVDs, MovieTypes, and Formats tables. The retrieved data is inserted in the DVDs2 table. When data is added to a table in this way, it is inserted after the existing data.

Importing Data into a Table

In addition to allowing you to export and copy data, MySQL allows you to import data from text files. To import data, you can use the mysql client utility or you can use the mysqlimport client utility. In this section, you learn how to use both tools to copy data into your database.

Using the mysql Utility to Import Data

The mysql client utility provides a number of options that you can use to import data into your database:

- **The `LOAD DATA` statement:** You can use the `LOAD DATA` statement at the mysql command prompt to import delimited values directly from a text file.
- **The `source` command:** You can use the `source` command at the mysql command prompt to run SQL statements and mysql commands that are saved in a text file. The statements can include `INSERT` statements that define values to be added to your tables.
- **The `mysql` command:** You can use the `mysql` command at your operating system's command prompt to run SQL statements and mysql commands that are saved in a text file. The `mysql` command is the same command that you use to launch the mysql client utility. You can also use the command to execute statements in a text file, without actually launching the utility, and these statements can include `INSERT` statements that define values to be added to your tables.

Now that you know the basics, take a closer look at each one of these options.

Using the LOAD DATA Statement to Import Data

Of the three options available for using the mysql client utility to import data, the `LOAD DATA` statement is the only one that allows you to import values directly from a delimited text file. (These are the type of files that are created when you export data from a MySQL database.) The following syntax describes how a `LOAD DATA` statement is created:

```
<load data statement>::=
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE '<filename>'
[REPLACE | IGNORE]
INTO TABLE <table name>
[<import option> [<import option>]]
[IGNORE <number> LINES]
[(<column name> [{, <column name>}...])]
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

```
<import option>::=
{FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']}
| {LINES
   [STARTING BY '<value>']
   [TERMINATED BY '<value>']}
```

As you can see, the LOAD DATA statement includes a number of elements; however, relatively few of these elements are actually required. The following syntax shows the bare-bones components that make up a basic LOAD DATA statement:

```
LOAD DATA INFILE '<filename>'
INTO TABLE <table name>
```

At the very least, you must specify two clauses: the LOAD DATA clause, which takes a filename as an argument, and the INTO TABLE clause, which takes a table name as an argument. The syntax demonstrates how simple the LOAD DATA statement can be. The statement can include a number of optional elements, however, so take a look at each line of the full syntax so that you have a complete picture of how the LOAD DATA statement works.

The first line of syntax is as follows:

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE '<filename>'
```

As the syntax shows, you must start with the LOAD DATA keywords, which introduce the LOAD DATA clause. The keywords are followed by the LOW_PRIORITY and CONCURRENT options. You can specify only one of the options. Use LOW_PRIORTY if you want to wait to load the data into the table until no clients are reading from the table. Use CONCURRENT (for MyISAM tables only) if you want to permit clients to retrieve data while rows are being added to the table.

By default, the LOAD DATA statement retrieves data from files stored on the MySQL server host. If you are accessing MySQL from that host, then the file is read from the local location. If you're accessing MySQL from a client computer and the target file is located on that computer, you can specify the LOCAL keyword to direct MySQL to look for the file on the client computer rather than the server host.

Once you specify any option that you want to include in the LOAD DATA clause, you must specify the INFILE keyword, followed by the name of the target file, enclosed in single quotes. If only a filename is specified, and no directory path, MySQL looks for the file in the folder associated with the current database (which is located in the data directory). You can also specify a full path, in which case MySQL looks in the specified location for the file.

The next set of options that you can add to your LOAD DATA statement is shown in the following syntax:

```
[REPLACE | IGNORE]
```

The two options refer to values that are duplicated in a unique index when you try to insert data. If neither option is specified, you receive an error if you try to insert a row in which a value in a unique index would be duplicated. You can override this behavior by using one of the two options. Use the REPLACE option if you want to replace the existing rows with new rows. Use the IGNORE option if you do not want to replace the existing rows, but you do want the insert process to continue.

If you specify the LOCAL *option, file transmission cannot be interrupted. As a result, the default behavior is the same as if the* IGNORE *option were specified.*

Next in the LOAD DATA syntax is the following INTO TABLE clause:

```
INTO TABLE <table name>
```

As you can see, this clause requires only that you specify the name of the table to which data will be added. After the INTO TABLE clause, you can specify one or two import options, as shown in the following syntax:

```
[<import option> [<import option>]]

<import option>::=
{FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']}
| {LINES
   [STARTING BY '<value>']
   [TERMINATED BY '<value>']}
```

No doubt, much of this syntax looks familiar to you. The import options of the LOAD DATA statement work just like the export options in a SELECT statement. You can specify a FIELDS clause, a LINES clause, or both. If you specify both, you must specify the FIELDS clause before the LINES clause. In addition, you must include at least one subclause for each clause you include.

The subclauses in the FIELDS and LINES clauses use the same defaults as are used when exporting data. As a result, if you use the default values when you export a file, you can use the default values when you import a file. If you're importing files that are generated outside MySQL, you can use the FIELDS and LINES subclauses as appropriate to match how the target files have been formatted. (Refer back to the section "Exporting Data to an Out File" for more information on the FIELDS and LINES clauses and their subclauses.)

For text files created in some Windows applications, you might find that you have to specify the LINES TERMINATED BY '\r\n' *subclause in a* LOAD DATA *statement in which you would normally use the default setting because of the way that those applications handle new lines.*

The next optional component of the LOAD DATA statement is the IGNORE clause, which is shown in the following syntax:

```
[IGNORE <number> LINES]
```

The IGNORE clause allows you to specify that a certain number of rows are ignored when the values are added to the table. The rows discarded are the first ones to be returned by the LOAD DATA statement. For example, if you specify IGNORE 10 LINES, the first 10 rows are ignored, and the rest of the data is added to the table.

TEAM LinG - Live, Informative, Non-cost and Genuine !

After the IGNORE clause, you can specify one or more columns from the target table, as shown in the following syntax:

```
[(<column name> [{, <column name>}...])]
```

As you can see, if more than one column name is included, they must be separated by commas. If you specify column names, each row returned by the LOAD DATA statement must include one value for each specified column. If you don't specify any column names, each row returned must include one value for each column in the table.

Now take a look at a couple of examples of how the LOAD DATA statement works. These examples are based on the CDs table used in earlier examples and the CDs3 table, which is shown in the following table definition:

```
CREATE TABLE CDs3
(
   CDName VARCHAR(50) NOT NULL,
   InStock SMALLINT UNSIGNED NOT NULL,
   Category VARCHAR(20)
);
```

For the purposes of these examples, assume that data has been exported out of the CDs table by using the following SELECT statement:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsBlues.sql'
FROM CDs WHERE Category='Blues';
```

Now suppose that you want to import data from the CDsBlues.sql file. To do so, you can use the following LOAD DATA statement:

```
LOAD DATA INFILE 'CDsBlues.sql'
INTO TABLE CDs3;
```

As you can see, the statement includes only the required components. The LOAD DATA clause includes the INFILE keyword and the filename, and the INTO TABLE clause includes the table name. Once you execute this statement, you can use the following SELECT statement to view the contents of the CDs3 table:

```
SELECT * FROM CDs3;
```

The statement should return a result set similar to the following:

```
+----------------------+---------+----------+
| CDName               | InStock | Category |
+----------------------+---------+----------+
| Mississippi Blues    |       2 | Blues    |
| Richland Woman Blues |      22 | Blues    |
| Stages               |      42 | Blues    |
+----------------------+---------+----------+
3 rows in set (0.00 sec)
```

The CDs3 table now contains the data that was first exported to the CDsBlues.sql file and then imported from the file.

Be sure that each row in the file that contains the data to be imported includes the correct number of values. The number of values should match the number of columns, and the values should be of a type compatible with the columns. If you try to insert too few or too many values per row, you will receive undesired results. MySQL attempts to insert the data in the order that it appears in the files. For example, if your file includes three values per row and your table includes five columns, MySQL attempts to place the first three values in the first three columns in the order that the values are specified, and no values are inserted in the remaining two columns. If a value cannot be inserted in a targeted column, the value is truncated or ignored.

The next example to examine uses data from the CDsCountry.sql file, which has been created by using the following `SELECT` statement:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDsCountry.sql'
   FIELDS
      TERMINATED BY ','
      ENCLOSED BY '"'
FROM CDs WHERE Category='Country';
```

Notice that the values are separated by a comma and enclosed by double quotes. As a result, you need to specify these characters when you import the data, as shown in the following `LOAD DATA` statement:

```
LOAD DATA INFILE 'CDsCountry.sql'
INTO TABLE CDs3
FIELDS
   TERMINATED BY ','
   ENCLOSED BY '"';
```

The statement contains the same required elements contained in the previous `LOAD DATA` example; however, the new statement also contains a `FIELDS` clause that specifies two subclauses. The subclauses are used because the values in the targeted file are separated with commas and enclosed in double quotes.

If, after running the `LOAD DATA` statement, you were to use a `SELECT` statement to retrieve all the content that is now contained in the CDs3 table, you would see a result set similar to the following:

```
+----------------------+---------+----------+
| CDName               | InStock | Category |
+----------------------+---------+----------+
| Mississippi Blues    |       2 | Blues    |
| Richland Woman Blues |      22 | Blues    |
| Stages               |      42 | Blues    |
| Ain't Ever Satisfied |      23 | Country  |
| Mud on the Tires     |      12 | Country  |
+----------------------+---------+----------+
5 rows in set (0.01 sec)
```

As you can see, two new rows have been added to the CDs3 table.

This Try It Out exercise has you import data into the DVDs2 table that you created in an earlier Try It Out section. You import data from files that you created when you exported data from the DVDRentals database.

TEAM LinG - Live, Informative, Non-cost and Genuine !

Try It Out Using the LOAD DATA Statement to Import Data into the DVDRentals Database

The following steps describe how to import the data:

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

```
use DVDRentals
```

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

2. First, you should remove the data currently stored in the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
TRUNCATE DVDs2;
```

   You should receive a message indicating that your query executed successfully, affecting no rows.

3. Now import data from the AvailDVDs.txt file into the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
LOAD DATA INFILE 'AvailDVDs.txt'
INTO TABLE DVDs2;
```

   You should receive a message indicating that your query executed successfully, affecting five rows.

4. Your next step is to view the contents of the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
SELECT * from DVDs2;
```

   You should receive results similar to the following:

```
+--------------------------------+--------------+-------------+----------+
| DVDName                        | MTypeDescrip | FormDescrip | RatingID |
+--------------------------------+--------------+-------------+----------+
| Amadeus                        | Drama        | Widescreen  | PG       |
| Mash                           | Comedy       | Widescreen  | R        |
| The Maltese Falcon             | Drama        | Widescreen  | NR       |
| The Rocky Horror Picture Show  | Comedy       | Widescreen  | NR       |
| What's Up, Doc?                | Comedy       | Widescreen  | G        |
+--------------------------------+--------------+-------------+----------+
5 rows in set (0.00 sec)
```

5. In order to try out another `LOAD DATA` statement, you should delete the records from the DVDs2 table once more. Execute the following SQL statement at the mysql command prompt:

```
TRUNCATE DVDs2;
```

   You should receive a message indicating that your query executed successfully, affecting no rows.

6. Next, import data from the AvailDVDs2.txt file into the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
LOAD DATA INFILE 'AvailDVDs2.txt'
INTO TABLE DVDs2
FIELDS TERMINATED BY '*,*'
LINES TERMINATED BY '**\n';
```

   You should receive a message indicating that your query executed successfully, affecting five rows.

7. Once more, you should view the contents of the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
SELECT * from DVDs2;
```

   You should receive the same results as you received in Step 4.

How It Works

The first `LOAD DATA` statement that you created includes only the required elements:

```
LOAD DATA INFILE 'AvailDVDs.txt'
INTO TABLE DVDs2;
```

The `LOAD DATA` clause includes the `INFILE` keyword and the name of the file (`AvailDVDs.txt`) that contains the data to be imported. The `INTO TABLE` clause contains the name of the table (`DVDs2`) in which data from the file is added. Because data had been exported to the file by using default formatting values, no `FIELDS` or `LINES` clause is required. The next statement that you created, however, required both clauses:

```
LOAD DATA INFILE 'AvailDVDs2.txt'
INTO TABLE DVDs2
FIELDS TERMINATED BY '*,*'
LINES TERMINATED BY '**\n';
```

The statement includes the required elements plus the optional `FIELDS` and `LINES` clauses. The `FIELDS TERMINATED BY` subclause specifies that the values in each row (in the `AvailDVDs2.txt` file) are separated by an asterisk, a comma, and another asterisk (`*,*`). The `LINES TERMINATED BY` subclause specifies that the rows are terminated by double asterisks and a newline symbol (`**\n`).

Using the source Command to Import Data

You can use the `source` command at the mysql command prompt to run SQL statements and commands that are stored in a text file. To use the `source` command, you need only to specify the path and filename of the file that contains the statements and command. For example, assume that you have created a file named CDsJazz.sql and added the following SQL statement to the file:

```
INSERT INTO CDs3
VALUES ('New Orleans Jazz', 17, 'Jazz'),
('Live in Paris', 18, 'Jazz');
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

As you can see, the file adds values to the CDs3 table, which is the table used in the examples in the last section. You can specify multiple statements and commands in a file, as long as each statement and command is terminated by a semi-colon so that MySQL knows where one ends and the other begins.

Once your text file is created, you can use the `source` command to run the statements and commands in the file, as shown in the following example:

```
source c:\program files\mysql\mysql server 4.1\data\test\CDsJazz.sql
```

As you can see, you need to specify only the `source` command and the path and filename. When you run this command, the SQL statements and MySQL commands in the file are executed. You can also use the following convention to use the source command:

```
\. c:\program files\mysql\mysql server 4.1\data\test\CDsJazz.sql
```

In this case, the backslash and period (`\.`) replace the word source, but the path and filename are still specified. Once you run the `source` command, you can verify the results by using a `SELECT` statement to retrieve all the contents from the CDs3 table, which should be similar to the following:

```
+----------------------+---------+----------+
| CDName               | InStock | Category |
+----------------------+---------+----------+
| Mississippi Blues    |       2 | Blues    |
| Richland Woman Blues |      22 | Blues    |
| Stages               |      42 | Blues    |
| Ain't Ever Satisfied |      23 | Country  |
| Mud on the Tires     |      12 | Country  |
| Live in Paris        |      18 | Jazz     |
| New Orleans Jazz     |      17 | Jazz     |
+----------------------+---------+----------+
7 rows in set (0.00 sec)
```

As you can see, the new rows have been added to the CDs3 table.

In the following exercise, you try out the `source` command to add data to the DVDs2 table that you created in an earlier Try It Out section.

Try It Out Using the source Command to Import Data into the DVDRentals Database

To try out the `source` command, you should first create a text file that contains the necessary `INSERT` statement. The following steps describe how to test the `source` command:

1. Create a text file in a text editor (such as Notepad) and name it DVD_1.sql. Add the following statement to the file, and save the file in the DVDRentals folder of your data directory:

```
INSERT INTO DVDs2
VALUES ('A Room with a View', 'Drama', 'Widescreen', 'NR');
```

TEAM LinG - Live, Informative, Non-cost and Genuine !

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

```
use DVDRentals
```

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

3. Next, use the `source` command to execute the `INSERT` statement in the DVD_1.sql file. Execute the following command at the mysql command prompt:

```
source <path>DVD_1.sql
```

The *<path>* placeholder refers to the DVDRentals folder in your data directory, which is where the text file should be stored. Be sure to enter the entire path, followed by the filename. When you execute the command, you should receive a message indicating that the command executed successfully and that one row was affected.

4. Now you should view the contents of the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
SELECT * FROM DVDs2;
```

You should receive results similar to the following:

```
+--------------------------------+-------------+------------+----------+
| DVDName                        | MTypeDescrip | FormDescrip | RatingID |
+--------------------------------+-------------+------------+----------+
| Amadeus                        | Drama        | Widescreen  | PG       |
| Mash                           | Comedy       | Widescreen  | R        |
| The Maltese Falcon             | Drama        | Widescreen  | NR       |
| The Rocky Horror Picture Show  | Comedy       | Widescreen  | NR       |
| What's Up, Doc?                | Comedy       | Widescreen  | G        |
| A Room with a View             | Drama        | Widescreen  | NR       |
+--------------------------------+-------------+------------+----------+
6 rows in set (0.00 sec)
```

How It Works

After you created a text file that contains an `INSERT` statement, you used the following `source` command to retrieve data from that file:

```
source <path>DVD_1.sql
```

As you can see, the command required only the `source` keyword, followed by a path and filename. When you ran this command, the `INSERT` statement in the DVD_1.sql file was executed and data was added to the DVDs2 table.

Using the mysql Command to Import Data

You can also use the `mysql` command at your operating system's command prompt to execute SQL statements and mysql commands in a text file. To do so, you must enter the `mysql` command, followed by the name of the database, the less than (<) symbol, and the path and filename of the file that contains the statements and command.

Take a look at an example to demonstrate how this works. For this example, assume that a file named CDsRock.sql has been created and that the following `INSERT` statement has been added to the file.

TEAM LinG - Live, Informative, Non-cost and Genuine !

```
INSERT INTO CDs3
VALUES ('Bloodshot', 10, 'Rock');
```

You can now exit the mysql client utility and execute that statement from your operating system's command prompt. For example, the following command executes the `INSERT` statement in the CDsRock.sql file:

```
mysql test < "c:\program files\mysql\mysql server 4.1\data\test\CDsRock.sql"
```

When specifying a path and filename in a MySQL command at your operating system's command prompt, you must enclose the path and filename in double quotes if either name contains spaces. For example, in the preceding command, the pathname includes spaces in the Program Files directory and the MySQL Server 4.1 directory.

As the statement shows, you must first specify the `mysql` command and then specify the database (test), the less than (<) symbol, and the path and filename (`c:\program files\mysql\mysql server 4.1\data\test\CDsRock.sql`). Once you run the `mysql` command statement, you can run a `SELECT` statement to retrieve the contents of the CDs3 table in order to verify that the command executed successfully. Your `SELECT` statement should return a result set similar to the following:

```
+----------------------+---------+----------+
| CDName               | InStock | Category |
+----------------------+---------+----------+
| Mississippi Blues    |       2 | Blues    |
| Richland Woman Blues |      22 | Blues    |
| Stages               |      42 | Blues    |
| Ain't Ever Satisfied |      23 | Country  |
| Mud on the Tires     |      12 | Country  |
| Live in Paris        |      18 | Jazz     |
| New Orleans Jazz     |      17 | Jazz     |
| Bloodshot            |      10 | Rock     |
+----------------------+---------+----------+
8 rows in set (0.00 sec)
```

Notice that the new row has been added to the CDs3 table.

Now that you've learned how to import data at your operating system's command prompt, you can try it out. In this exercise, you create a text file that includes an `INSERT` statement; then you use the `mysql` command to run that statement.

Try It Out Using the mysql Command to Import Data into the DVDRentals Database

Follow these steps to test the `mysql` command:

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

```
use DVDRentals
```

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

2. Create a text file in a text editor (such as Notepad), and name it DVD_2.sql. Add the following statement to the file, and save the file in the DVDRentals folder of your data directory:

```
INSERT INTO DVDs2
VALUES ('Out of Africa', 'Drama', 'Widescreen', 'PG');
```

3. At your operating system's command prompt, execute the following command:

```
mysql dvdrentals < <path>DVD_2.sql
```

   The *<path>* placeholder refers to the DVDRentals folder in your data directory, which is where the text file should be stored. Be sure to enter the entire path, followed by the filename.

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

```
use DVDRentals
```

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

5. Next, view the contents of the DVDs2 table. Execute the following SQL statement at the mysql command prompt:

```
SELECT * FROM DVDs2;
```

   You should receive results similar to the following:

```
+-------------------------------+-------------+------------+----------+
| DVDName                       | MTypeDescrip | FormDescrip | RatingID |
+-------------------------------+-------------+------------+----------+
| Amadeus                       | Drama       | Widescreen | PG       |
| Mash                          | Comedy      | Widescreen | R        |
| The Maltese Falcon            | Drama       | Widescreen | NR       |
| The Rocky Horror Picture Show | Comedy      | Widescreen | NR       |
| What's Up, Doc?               | Comedy      | Widescreen | G        |
| A Room with a View            | Drama       | Widescreen | NR       |
| Out of Africa                 | Drama       | Widescreen | PG       |
+-------------------------------+-------------+------------+----------+
7 rows in set (0.00 sec)
```

How It Works

In this exercise, you first created a text file that includes an INSERT statement. The statement adds a row of data to the DVDs2 table in the DVDRentals database. Once the file was created, you executed the following mysql command statement:

```
mysql dvdrentals < <path>DVD_2.sql
```

In the command, you specified the mysql command, followed by the name of the DVDRentals database. After the database name, you added a less than (<) symbol and then the path and filename. When you executed the command, the INSERT statement in the DVD_2.sql file was executed and a row was added to the DVDs2 table.

TEAM LinG - Live, Informative, Non-cost and Genuine !

Using the mysqlimport Utility to Import Data

MySQL includes a utility that allows you to import delimited data in a text file, without having to launch the mysql client utility. The mysqlimport utility supports many of the same functions as the `LOAD DATA` statement. To use the mysqlimport utility, you must specify the `mysqlimport` command, followed by any options that you want to include. Next, you must specify the name of the database and the path and filename of the file that contains the data to be inserted.

The `mysqlimport` command statement does not include the name of the target table. Instead, the table is determined by the name of the file. MySQL assumes that the filename will be the same as the table name (not counting any file extensions). For example, if you're inserting data in the Authors table, the file must be named Authors.txt, Authors.sql, or something similar to that.

Now take a look at a couple of examples that demonstrate how this works. The first example imports data from a file named CDs3.sql, which has been created by using the following `SELECT` statement:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDs3.sql'
FROM CDs WHERE Category='New Age';
```

Notice that the name of the file, not including the extension, is CDs3. As a result, you can insert data only in a table named CDs3, as shown in the following example:

```
mysqlimport --user=root --password=pw1 test "c:\program files\mysql\mysql server
4.1\data\test\CDs3.sql"
```

In this command statement (which is run from your operating system's command prompt), the `mysqlimport` command is specified, followed by the optional `--user` and `--password` arguments. After the arguments comes the name of the table (`test`) and the path and filename (`c:\program files\mysql\mysql server 4.1\data\test\CDs3.sql`). The user and password that are provided in the command statement are those that you would normally use when you access the MySQL databases.

You can verify whether the data has been successfully added to the table by starting the mysql client utility and running a `SELECT` statement that retrieves the contents of the CDs3 table. The `SELECT` statement should retrieve a result set similar to the following:

```
+----------------------+---------+----------+
| CDName               | InStock | Category |
+----------------------+---------+----------+
| Mississippi Blues    |       2 | Blues    |
| Richland Woman Blues |      22 | Blues    |
| Stages               |      42 | Blues    |
| Ain't Ever Satisfied |      23 | Country  |
| Mud on the Tires     |      12 | Country  |
| Live in Paris        |      18 | Jazz     |
| New Orleans Jazz     |      17 | Jazz     |
| Bloodshot            |      10 | Rock     |
| The Essence          |       5 | New Age  |
+----------------------+---------+----------+
9 rows in set (0.00 sec)
```

As the result set shows, a New Age row has been added to the table.

TEAM LinG - Live, Informative, Non-cost and Genuine !

Now take a look at another example. This time, when the CDs3 file is created, a `FIELDS` clause is included in the export definition:

```
SELECT CDName, InStock, Category INTO OUTFILE 'CDs3.sql'
   FIELDS
      TERMINATED BY ','
      ENCLOSED BY '*'
FROM CDs WHERE Category='Classical';
```

Remember, you cannot create an out file that already exists. If one does exist, rename the old file or delete it.

As the statement shows, the values are now separated by a comma and enclosed by asterisks. As a result, when you run the `mysqlimport` command, you must specify these formatting characters in the command statement, as shown in this example:

```
mysqlimport --user=root --password=pw1 --fields-terminated-by="," --fields-
enclosed-by="*" test "c:\program files\mysql\mysql server 4.1\data\test\CDs3.sql"
```

As you can see, two more arguments have been added to the command statement: `--fields-terminated-by=","` and `--fields-enclosed-by="*"`. The `mysqlimport` utility supports arguments that are counterparts to the `FIELDS` and `LINES` subclauses of a `LOAD DATA` statement. For example, the `FIELDS TERMINATED BY` subclause in the `LOAD DATA` statement is the same as the `--fields-terminated-by` argument in the mysql import utility. Each of the `FIELDS` and `LINES` subclauses follows the same format. (For details on all the `mysqlimport` arguments, see the MySQL product documentation.)

Now if you were to retrieve data from the CDs3 table, your result set would include Classical CDs, so you should receive results similar to the following:

```
+------------------------+---------+-----------+
| CDName                 | InStock | Category  |
+------------------------+---------+-----------+
| Mississippi Blues      |       2 | Blues     |
| Richland Woman Blues   |      22 | Blues     |
| Stages                 |      42 | Blues     |
| Ain't Ever Satisfied   |      23 | Country   |
| Mud on the Tires       |      12 | Country   |
| Live in Paris          |      18 | Jazz      |
| New Orleans Jazz       |      17 | Jazz      |
| Bloodshot              |      10 | Rock      |
| The Essence            |       5 | New Age   |
| Music for Ballet Class |       9 | Classical |
| The Magic of Satie     |      42 | Classical |
+------------------------+---------+-----------+
11 rows in set (0.00 sec)
```

In this Try It Out exercise, you use a `SELECT` statement to export data to the DVDs2.txt file. You then use the `mysqlimport` command to add data to the DVDs2 table.

TEAM LinG - Live, Informative, Non-cost and Genuine !

Try It Out Using the mysqlimport Utility to Import Data into the DVDRentals Database

The following steps describe how to carry out these tasks:

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

```
use DVDRentals
```

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

2. Create a text file in a text editor (such as Notepad) and name it DVDs2.txt. Add the following statement to the file and save the file in the DVDRentals folder of your data directory:

```
SELECT DVDName, MTypeDescrip, FormDescrip, d.RatingID
   INTO OUTFILE 'DVDs2.txt'
FROM DVDs AS d, MovieTypes AS m, Formats AS f
WHERE d.MTypeID=m.MTypeID AND d.FormID=f.FormID
   AND DVDID=1;
```

You should receive a message indicating that the statement executed successfully, affecting one row.

3. Type the following command at the mysql command prompt, and press Enter:

```
exit
```

You are returned to your operating system's command prompt.

4. At your operating system's command prompt, execute the following command:

```
mysqlimport --user=root --password=pw1 dvdrentals <path>DVDs2.txt
```

The *<path>* placeholder refers to the DVDRentals folder in your data directory, which is where the text file should be stored. Be sure to enter the entire path, followed by the filename. If your pathname includes spaces, be sure to enclose the entire path and filename in double quotes. You should receive a message indicating that the one record in the DVDs2 table of the DVDRentals database has been affected.

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

```
use DVDRentals
```

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

6. Now view the contents of the DVDs2 table to verify that the data has been added. Execute the following SQL statement at the mysql command prompt:

```
SELECT * FROM DVDs2;
```

You should receive results similar to the following:

```
+-------------------------------+--------------+-------------+----------+
| DVDName                       | MTypeDescrip | FormDescrip | RatingID |
+-------------------------------+--------------+-------------+----------+
| Amadeus                       | Drama        | Widescreen  | PG       |
| Mash                          | Comedy       | Widescreen  | R        |
| The Maltese Falcon            | Drama        | Widescreen  | NR       |
```

TEAM LING - LIVE, INFORMATIVE, NON-COST AND GENUINE !

```
| The Rocky Horror Picture Show | Comedy       | Widescreen  | NR       |
| What's Up, Doc?               | Comedy       | Widescreen  | G        |
| A Room with a View            | Drama        | Widescreen  | NR       |
| Out of Africa                 | Drama        | Widescreen  | PG       |
| White Christmas               | Musical      | Widescreen  | NR       |
+-------------------------------+--------------+-------------+----------+
8 rows in set (0.00 sec)
```

7. Finally, you should drop the DVDs2 table from the DVDRentals database. Execute the following SQL statement at the mysql command prompt:

```
DROP TABLE DVDs2;
```

You should receive a message indicating that the query executed successfully and that no rows were affected.

How It Works

In this exercise, you used a `SELECT` statement to join tables in the DVDRentals database and then export data retrieved by the statement to a file named DVDs2.sql. You then used the mysqlimport utility to import data from the file into the DVDs2 table, as shown in the following statement:

```
mysqlimport --user=root --password=pw1 dvdrentals <path>DVDs2.txt
```

In this command statement, you specified the `mysqlimport` command and the `--user` and `--password` arguments. The user (`root`) and password (`pw1`) are the user and password that you use to launch the mysql client utility. (These are the values that should be stored in your configuration file.) After you defined the two arguments, you specified the name of the DVDRentals database and the path and filename that contained the data to be imported. When you ran the `mysqlimport` command, the values in the DVDs2.sql file were added to the DVDs2 table. MySQL inserted the values in the table with the same name as the file.

Summary

As the chapter has demonstrated, MySQL provides a number of SQL statements and commands that allow you to export, copy, and import data in a database. As a result, you can work with large quantities of data that can be copied between tables in the database or copied to or from files outside the database. In this chapter, you learned how to use the following SQL statements and MySQL commands to export, copy, and import data:

- The `SELECT` statement, in order to export data into out files and dump files
- The `CREATE TABLE` statement, in order to copy data into a new table as you're creating that table
- The `INSERT` statement, in order to copy data into an existing table
- The `REPLACE` statement, in order to copy data into an existing table
- The `LOAD DATA` statement, in order to import data into a table
- The `source`, `mysql`, and `mysqlimport` commands, in order to import data into a table

TEAM LinG - Live, Informative, Non-cost and Genuine !

Being able to import and export data is particularly useful when working with other applications or database products that require the exchange of data. For example, suppose that you want to import data from a database other than MySQL. To do so, you can simply export the data out of the other database into text files and then import the data into the MySQL database from the text files. Nearly any data that can be saved to text files and clearly delimited can be imported into MySQL, and any application or database product that can import data from a text file can import data from MySQL. As a result, you can easily manage large quantities of data from different data sources. In Chapter 12, you learn how to manage transactions to ensure safe data access by multiple users, as well as how to use transactions to manage the execution of SQL statements.

Exercises

For these exercises, you use SQL statements and MySQL commands to export, copy, and import data. The statements and commands are based on the Produce table in the test database. The Produce table is based on the following table definition:

```
CREATE TABLE Produce
(
   ProdID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ProdName VARCHAR(40) NOT NULL,
   Variety VARCHAR(40) NULL,
   InStock SMALLINT UNSIGNED NOT NULL
);
```

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

```
INSERT INTO Produce
VALUES (101, 'Apples', 'Red Delicious', 2000),
(102, 'Apples', 'Fuji', 1500),
(103, 'Apples', 'Golden Delicious', 500),
(104, 'Apples', 'Granny Smith', 300),
(105, 'Oranges', 'Valencia', 1200),
(106, 'Oranges', 'Seville', 1300),
(107, 'Cherries', 'Bing', 2500),
(108, 'Cherries', 'Rainier', 1500),
(109, 'Mushrooms', 'Shitake', 800),
(110, 'Mushrooms', 'Porcini', 400),
(111, 'Mushrooms', 'Portobello', 900);
```

Use the Produce table to complete the following exercises. You can find the answers to these exercises in Appendix A.

1. Create an SQL statement that exports data from the Produce table to a text file named Apples.txt. The file should be saved to the folder associated with the test database. The rows exported should include only the ProdName, Variety, and InStock columns and only those rows that contain a ProdName value of Apples. The exported data should be copied to the text file in a default format.

2. Create an SQL statement that exports data from the Produce table to a text file named Oranges.txt. The file should be saved to the folder associated with the test database. The rows exported should include only the ProdName, Variety, and InStock columns and only those rows that contain a ProdName value of Oranges. The exported data should be copied to the text file in a default format, except that the fields should be terminated by a comma (,) and should be enclosed by an asterisk (*).
3. Create an SQL statement that creates a table named Produce2. The table should be identical to the Produce table, except that it should not include the ProdID column. When creating the table, copy data from the Produce table to the Produce2 table. The copied data should include only the ProdName, Variety, and InStock columns and only those rows that contain a ProdName value of Cherries.
4. Create an SQL statement that adds data to the Produce2 table that you created in Step 3. The data should be made up of ProdName, Variety, and InStock values from the Produce table. In addition, the data should include only those rows that contain a ProdName value of Mushrooms.
5. Create an SQL statement that adds data to the Produce2 table that you created in Step 3. The data should be retrieved from the Apples.txt file that you created in Step 1.
6. Create an SQL statement that adds data to the Produce2 table that you created in Step 3. The data should be retrieved from the Oranges.txt file that you created in Step 2.

TEAM LinG - Live, Informative, Non-cost and Genuine !