5

Managing Databases, Tables, and Indexes

In the first four chapters of the book, you were provided with the information necessary to install and use MySQL and design relational databases that could be implemented in MySQL. You were also provided with a foundation in the principles of SQL and the relational model. From all this information, you should now have the background you need to begin creating databases and adding the objects that those databases should contain.

The first step in setting up a MySQL database is to create the actual database object, which serves as a container for the tables in that database. The database acts as a central point of administration for the tables in the database. The actual data is stored in the tables, which provide a structured organization for the data and maintain the integrity of that data. Associated with each table is a set of indexes that facilitate access to the data in the tables.

In this chapter, you learn how to create and manage the database object, the tables in the database, and the indexes associated with the tables. To provide you with the information necessary to perform all these tasks, the chapter covers the following topics:

❑ How to create a database and specify a character set and collation name for that database. You also learn how to modify database properties and remove a database from your system.

❑ The information necessary to create a table, define columns in that table, add constraints to the table, create indexes on the table, and specify the type of table to be created.

❑ Information about the various types of indexes that you can add to a table, how to add indexes to a table, and how to remove an index from a table.

❑ How to retrieve information about databases and tables so that you can see what databases and tables exist and how those databases and tables are configured.

Managing Databases

Once you install MySQL and are comfortable using the tools in MySQL — particularly the mysql client utility — you can begin creating databases in the MySQL environment. Recall from Chapter 4

that when you add a database to MySQL, a corresponding subdirectory is automatically added to the data directory. Any tables added to the database appear as files in the subdirectory. If you then remove a database, the corresponding subdirectory and files are also deleted.

As you can see, the first step in setting up a database in MySQL is to create the database object and, subsequently, the corresponding subdirectories. From there, you can modify the database definition or delete the database. This section discusses each of these tasks.

Creating Databases

Creating a database in MySQL is one of the easier tasks that you're likely to perform. At its most basic, a database definition statement requires only the keywords CREATE DATABASE, followed by the name of the new database, as shown in the following syntax:

```
<database definition>::=
CREATE DATABASE [IF NOT EXISTS] <database name>
[[DEFAULT] CHARACTER SET <character set name>]
[[DEFAULT] COLLATE <collation name>]
```

As the syntax shows, very few components of the CREATE DATABASE statement are actually required. The statement does include several optional elements. The first of these — IF NOT EXISTS — determines how MySQL responds to a CREATE DATABASE statement if a database with the same name already exists. If a database already exists and the IF NOT EXISTS clause is not used, MySQL returns an error. If the clause is used, MySQL returns a warning, without returning any errors. Regardless of whether the clause is included in the statement, the effect on the database is the same: If a database with the same name exists, no new database is created.

The next optional components of the CREATE DATABASE statement are the CHARACTER SET clause and the COLLATE clause. You can specify either one of the two clauses or both. (The DEFAULT keyword is optional in either case and has no effect on the outcome.) The CHARACTER SET clause specifies the default character set to use for a new database, and the COLLATE clause specifies which default collation to use. A *character set* is a collection of letters, numbers, and symbols that create the values in a database. For example, A, B, C, a, b, c, 1, 2, 3, >, +, and * are all part of a character set. A *collation* is a named sorting order used for a specified character set. Collations define the way in which values made up of a particular character set are compared, sorted, and grouped together. Most character sets have one or more collations associated with them. For example, some of the collations associated with the default character set, latin1, include latin1_bin, latin1_general_ci, and latin1_swedish_ci, which is the default collation. If you do not specify the CHARACTER SET or COLLATION clause, the database uses the default MySQL character set or collation.

> You can view the character sets and collations available in your system by executing the SHOW CHAR-ACTER SET *and* SHOW COLLATION *statements in the mysql client utility. Also note that character sets and collations affect only string data (letters, numbers, and symbols), as opposed to all numerical data or data related to dates and times.*

Now that you have an understanding of the syntax used to create a database, take a look at a couple examples. The first example is a basic CREATE DATABASE statement that includes no optional components:

```
CREATE DATABASE BookSales;
```

When you execute the statement, a database named BookSales is added to your system. The database uses the default character set and collation because you specified neither.

When you create databases and tables in Windows, all names are converted to lowercase. Because Windows filenames and directory names are case insensitive, it follows that case is not a factor when specifying database and table names. In Linux and other Unix-like operating systems, the case is preserved. Because filenames and directory names are case sensitive, you must specify the correct case when referencing database and table names.

In the next example, the CREATE DATABASE statement specifies the character set and collation:

```
CREATE DATABASE BookSales
DEFAULT CHARACTER SET latin1
DEFAULT COLLATE latin1_bin;
```

In this example, the CHARACTER SET clause specifies the latin1 character set, and the COLLATE clause specifies the latin1_bin collation. In both cases, you use the DEFAULT keyword, but it isn't required. Executing the statement creates a database named BookSales, which uses the specified character set and collation.

In the Try It Out sections in Chapter 4, you created a data model for the DVDRentals database. In the Try It Out examples in this chapter, however, you create the database based on that model. Later in the chapter, you create the tables in the DVDRentals database.

Try It Out   Creating the DVDRentals Database

To create the database, follow these steps:

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

```
CREATE DATABASE DVDRentals;
```

You should receive a message indicating that your statement executed successfully.

2. In order to create tables or execute any statements in the context of the new database, you should switch over to that new database by using the following command:

```
use DVDRentals
```

You should receive a message indicating the database change.

How It Works

In this exercise, you used a CREATE DATABASE statement to create the DVDRentals database. This is the database for which you created a data model in Chapter 4. Because you didn't specify any character set or collation, the database uses the default values, which for a standard MySQL installation are the character set latin1 and the collation latin1_swedish_ci. Once you create the DVDRentals database, you can begin adding the necessary tables.

Modifying Databases

There might be times when you want to change the character set or collation used for your database. To do this, you can use an ALTER DATABASE statement to specify the new settings. As you can see from the following syntax, the ALTER DATABASE statement is similar to the CREATE DATABASE statement:

```
ALTER DATABASE <database name>
[[DEFAULT] CHARACTER SET <character set name>]
[[DEFAULT] COLLATE <collation name>]
```

In this statement, you must specify the ALTER DATABASE keywords and the name of the database, along with the CHARACTER SET clause, the COLLATE clause, or both. For either clause, simply specify the name of the character set and collation. For example, to change the character set to latin1 for the BookSales database, use the following ALTER DATABASE statement:

```
ALTER DATABASE BookSales
CHARACTER SET latin1;
```

As you can see, the statement specifies only a CHARACTER SET clause, which means the current collation remains the same.

Use caution when changing the character set for a database. In some cases, changing the character set can result in the database no longer supporting all the characters stored as data in the database.

Deleting Databases

Deleting a database from your system is merely a matter of executing a DROP DATABASE statement. The following syntax shows the components that make up the statement:

```
DROP DATABASE [IF EXISTS] <database name>
```

The statement requires only the DROP DATABASE keywords and the name of the database. In addition, you can specify the IF EXISTS clause. If you specify this clause and a database with that name doesn't exist, you receive a warning message rather than an error. Now take a look at an example of the DROP DATABASE statement:

```
DROP DATABASE BookSales;
```

This example removes the BookSales database from the system. When you remove a database, you also remove the tables in that database and any data contained in the table. As a result, you want to be extremely cautious whenever you execute the DROP DATABASE command.

Managing Tables

The next step in setting up a database, after creating the actual database object, is to add the necessary tables to that database. The tables provide a structure for storing and securing the data. All data exists within the structure of the tables, and all tables exist within the structure of the database. In addition to creating tables, you can also modify the table definitions or delete the tables from the database. This section explains how to perform each of these tasks.

Creating Tables

To create a table in MySQL, you must use the CREATE TABLE statement to define the columns in the table and configure the appropriate constraints on the table. The CREATE TABLE statement is one of the most complex SQL statements in MySQL. It contains numerous components and provides many options for defining the exact nature of a particular table. The following syntax represents the elements that make up a CREATE TABLE statement:

```
<table definition>::=
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
(<table element> [{, <table element>}...])
[<table option> [<table option>...]]

<table element>::=
<column definition>
| {[CONSTRAINT <constraint name>] PRIMARY KEY
     (<column name> [{, <column name>}...])}
| {[CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
     (<column name> [{, <column name>}...]) <reference definition>}
| {[CONSTRAINT <constraint name>] UNIQUE [INDEX] [<index name>]
     (<column name> [{, <column name>}...])}
| {{INDEX | KEY} [<index name>] (<column name> [{, <column name>}...])}
| {FULLTEXT [INDEX] [<index name>] (<column name> [{, <column name>}...])}

<column definition>::=
<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]
[PRIMARY KEY] [COMMENT '<string>'] [<reference definition>]

<type>::=
<numeric data type>
| <string data type>
| <data/time data type>

<reference definition>::=
REFERENCES <table name> [(<column name> [{, <column name>}...])]
[ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
[ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
[MATCH FULL | MATCH PARTIAL]

<table option>::=
{ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MYISAM}}
| <additional table options>
```

As you can see, many elements make up a CREATE TABLE statement. In fact, the syntax shown here is not the CREATE TABLE statement in its entirety. As you move through the chapter, other elements are introduced and some elements, which are beyond the scope of this book, are not discussed at all. Still, this chapter covers all the essential components, so by the end of the chapter, you should have a fairly comprehensive foundation on how to create a table definition.

Now take a closer look at the CREATE TABLE syntax. The best place to begin is at the first section:

```
<table definition>::=
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
(<table element> [{, <table element>}...])
[<table option> [<table option>...]]
```

143

This section represents the entire CREATE TABLE statement, with individual components being explained later in the syntax. The first line of the actual statement requires only the keywords CREATE TABLE, followed by the name of the new table. This line also contains two optional components. The first — TEMPORARY — indicates that this is a temporary table used only during the current session by the current user. A temporary table exists only as long as the session is open or the table is explicitly dropped. The second optional element is the IF NOT EXISTS clause. You've seen this clause before in the CREATE DATABASE statement. When you include it in your CREATE TABLE statement, a warning is generated, rather than an error, if a table by the same name already exists when you execute this statement.

The next line of syntax allows you to define the individual table elements, as represented by the <table element> placeholder. A table element is an individual object that is defined on a table, such as a column or PRIMARY KEY constraint. Each CREATE TABLE statement includes one or more table elements. If more than one table element exists, they are separated by commas. Regardless of how many table elements exist, they are all enclosed in a set of parentheses.

The last line in the first section of syntax allows you to define individual table options. Table options are options that apply to the table as a whole. For example, you can define the type of table that you want to create. All table options are optional; however, you can define as many as are necessary.

As you can see from the first section of syntax, a CREATE TABLE statement can be as simple or as complicated as you need to make it. The only required elements are the CREATE TABLE clause and at least one table element enclosed in parentheses, as shown in the following syntax:

```
CREATE TABLE <table name> (<table element>)
```

Because a table element is a required component, take a look at the next section of syntax:

```
<table element>::=
<column definition>
| {[CONSTRAINT <constraint name>] PRIMARY KEY
    (<column name> [{, <column name>}...])}
| {[CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
    (<column name> [{, <column name>}...]) <reference definition>}
| {[CONSTRAINT <constraint name>] UNIQUE [INDEX] [<index name>]
    (<column name> [{, <column name>}...])}
| {{INDEX | KEY} [<index name>] (<column name> [{, <column name>}...])}
| {FULLTEXT [INDEX] [<index name>] (<column name> [{, <column name>}...])}
```

A table element can represent one of many different options. The most commonly used option is the one represented by the <column definition> placeholder, which, as the name suggests, allows you to define a column to include in your table definition. You are likely, though, to use the other options with regularity. For this reason, the following sections examine each of these options individually.

Creating Column Definitions

A column definition is one type of table element that you can define in a table definition. You must create a column definition for each column that you want to include in your table. The following syntax provides you with the structure that you should use when creating a column definition:

```
<column definition>::=
<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]
[PRIMARY KEY] [COMMENT '<string>'] [<reference definition>]
```

As you can see, only two elements are required in a column definition: the column name (represented by the `<column name>` placeholder) and the data type (represented by the `<type>` placeholder). The name can be any acceptable identifier in MySQL, and the database can be any of the supported data types. Each additional element of the column definition is optional and, along with data types, they are discussed individually in the following sections.

Defining Data Types

As you recall from earlier chapters, a data type is a constraint placed on a particular column to limit the type of values that you can store in that column. MySQL supports three categories of data types, as represented by the following syntax:

```
<type>::=
<numeric data type>
| <string data type>
| <data/time data type>
```

Whenever you add a column to your table, you must define the column with a data type that is in one of these three categories. Each category of types has its own specific characteristics, and each places restrictions on the type of data that you can include in a particular column. Take a closer look at each category to better understand the characteristics of each.

Numeric Data Types

As the name suggests, *numeric* data types are concerned with numbers. If you have a column that will contain nothing but numbers, chances are that you want to configure that column with a numeric data type.

You can divide numeric data types into two categories, as the following syntax suggests:

```
<numeric data type>::=
<integer data type> [(<length>)] [UNSIGNED] [ZEROFILL]
| <fractional data type> [(<length>, <decimals>)] [UNSIGNED] [ZEROFILL]
```

Each of the two categories of integer data types supports several options. The first of these is represented by the `<length>` placeholder, which indicates the maximum number of displayed characters for a particular column. You can specify that the length be anywhere from 1 to 255. The fractional data types also include the `<decimals>` placeholder. This value indicates the number of decimal places to be displayed for a value in the column. You can specify that the number of decimal places be anywhere from 0 to 30; however, `<decimals>` must always be at least two less than `<length>`.

The next option available for the numeric data types is `UNSIGNED`. When this option follows a numeric data type, no negative values are permitted in the column. If you specify `ZEROFILL`, zeros are added to the beginning of a value so that the value is displayed with the number of characters represented by the `<length>` placeholder. For example, if you define `<length>` as 4 and you specify `ZEROFILL`, the number 53 displays as 0053. `ZEROFILL` is most useful when used in conjunction with a specified length. In addition, `UNSIGNED` is assumed when you specify `ZEROFILL`, even if `UNSIGNED` isn't explicitly specified. In other words, you can't use `ZEROFILL` for negative numbers.

Now take a look at the two categories of numeric data types. The first of these is the *integer* data type. Integer data types allow you to store only whole numbers in your column (no fractions or decimals). MySQL supports the integer data types shown in the following syntax:

```
<integer data type>::=
  TINYINT | SMALLINT | MEDIUMINT | INT | INTEGER | BIGINT
```

Each of these data types specifies not only that whole numbers must be stored in the column, but also that the numbers stored must fall within a specific range of values, as shown in the following table.

Data type	Acceptable values	Storage requirements
TINYINT	Signed: –128 to 127 Unsigned: 0 to 255	1 byte
SMALLINT	Signed: –32768 to 32767 Unsigned: 0 to 65535	2 bytes
MEDIUMINT	Signed: –8388608 to 8388607 Unsigned: 0 to 16777215	3 bytes
INT	Signed: –2147483648 to 2147483647 Unsigned: 0 to 4294967295	4 bytes
INTEGER	Same values as the INT data type. (INTEGER is a synonym for INT.)	4 bytes
BIGINT	Signed: –9223372036854775808 to 9223372036854775807 Unsigned: 0 to 18446744073709551615	8 bytes

The range of acceptable values for each integer data type has nothing to do with the `<length>` placeholder. Whether you were to specify a length of 2 or 20, the stored value would be the same, as would be the value limitations. As the table demonstrates, signed values are different from unsigned values. If a column doesn't require negative values, using the UNSIGNED option increases the upper end of the range of stored values, although the storage requirements remain the same. For example, a value in a TINYINT column requires 1 byte of storage whether the column is signed or unsigned.

Now take a look at an example of a CREATE TABLE statement that includes two column definitions, one that uses the SMALLINT data type and one that uses the INT data type. The following statement creates a table named Inventory:

```
CREATE TABLE Inventory
(
    ProductID SMALLINT(4) UNSIGNED ZEROFILL,
    Quantity INT UNSIGNED
);
```

As you can see, the table includes a column named ProductID and a column named Quantity. The ProductID column is configured with a SMALLINT data type that specifies 4 as its maximum display length. Because a SMALLINT value can have a range of 0 to 65535, the display of 4 might not apply to all values. You would specify a display size smaller than the capacity only if you're certain that all digits will fall into that range. If your value does require the full five digits, they will all be displayed, despite the `<length>` value of 4. As a result, the only time including the length is useful is when you're also including the ZEROFILL option, which is the case in this column definition. As a result, no negative values are allowed and zeros are added to all values less than four characters wide.

The next column defined in this statement is the Quantity column. This column is defined with an `INT` data type, and it is also unsigned. As a result, negative numbers are not allowed in this column, which means that acceptable values can fall in the range of 0 to 4294967295.

Another thing to notice about this `CREATE TABLE` statement is that the column definitions are enclosed in parentheses and separated by a comma. All table elements, including column definitions, are treated in this manner.

Now take a look at the fractional data types, which are shown in the following syntax:

```
<fractional data type>::=
FLOAT | DOUBLE | DOUBLE PRECISION | REAL | DECIMAL | DEC | NUMERIC |FIXED
```

The *fractional* data types, unlike the integer data types, support the use of decimals. In fact, that is the key characteristic of these data types, which are described in the following table:

Data type	Description
FLOAT	An approximate numeric data type that uses 4 bytes of storage. The data type supports the following values: $-3.402823466E+38$ to $-1.175494351E-38$ 0 $1.175494351E-38$ to $3.402823466E+38$
DOUBLE	An approximate numeric data type that uses 8 bytes of storage. The data type supports the following values: $-1.7976931348623157E+308$ to $-2.2250738585072014E-308$ 0 $2.2250738585072014E-308$ to $1.7976931348623157E+308$
DOUBLE PRECISION	Synonym for the DOUBLE data type
REAL	Synonym for the DOUBLE data type
DECIMAL	An exact numeric data type whose range storage requirements depend on the <length> and <decimals> values specified in the column definition
DEC	Synonym for the DECIMAL data type
NUMERIC	Synonym for the DECIMAL data type
FIXED	Synonym for the DECIMAL data type

As described in the table, there are only three basic fractional data types: `FLOAT`, `DOUBLE`, and `DECIMAL`. The rest are synonyms for the `DOUBLE` and `DECIMAL` data types.

The `DOUBLE` data type supports a greater degree of precision than does the `FLOAT` data type. In other words, `DOUBLE` supports more decimal places than `FLOAT`. If you need the greater precision, you would use `DOUBLE`, although you should be aware that this doubles your storage requirements. In both cases,

values are stored as numeric data and are subject to errors caused by numbers being rounded, which is why they're referred to as *approximate* numeric types. Generally numbers are rounded according to the column <length> and <decimals> specifications, which can sometimes result in imprecise results.

The DECIMAL data type, which is referred to as an *exact* numeric data type, gets around the issue of round-off errors by storing the values as strings, with the <length> and <decimals> specifications determining storage requirements and range. You should use the DECIMAL data type when you require values to be completely accurate, such as when you're storing information about money. The drawback to using the DECIMAL data type is that there are trade-offs in performance compared to the approximate numeric types. For this reason, if you plan to store values that don't require the accuracy of the DECIMAL type, use FLOAT or DOUBLE.

Now take a look at a table definition that uses the DECIMAL and FLOAT data types. The following CREATE TABLE statement creates a table named Catalog:

```
CREATE TABLE Catalog
(
    ProductID SMALLINT,
    Price DECIMAL(7,2),
    Weight FLOAT(8,4)
);
```

As you can see, the table includes a DECIMAL column (named Price) and a FLOAT column (named Weight). The Price column contains a <length> value of 7 and a <decimals> value of 2. As a result, the values display with up to 7 characters and 2 decimal places, such as 46264.45 and 242.90.

Because of the storage requirements for DECIMAL values, positive DECIMAL values (as opposed to negative values) receive one extra character to display values. For example, a DECIMAL column that has a <length> value of 7 and a <decimals> value of 2 can actually have up to eight numeric characters — plus one character for the decimal point — for positive numbers, but only seven characters — plus one character for the negative sign and one for the decimal point — for negative numbers. FLOAT and DOUBLE values do not operate in the same way.

The second column in the preceding example is configured with the FLOAT data type. In this case, the total display can be eight characters long, with four characters to the right of the decimal point. The implication in this case is that the Weight value does not have to be as exact as the Price value, so you don't have to worry about errors caused by values that have been rounded.

String Data Types

The string data types provide a great deal of flexibility for storing various types of data from individual bits to large files. String data types are normally used to store names and titles and any value that can include letters and numbers. MySQL supports four categories of string data types, as shown in the following syntax:

```
<string data type>::=
<character data type>
| <binary data type>
| <text data type>
| <list data type>
```

The character data types are the ones that you probably use the most often. As the following syntax shows, there are two types of character data types:

```
<character data type>::=
CHAR (<length>) [BINARY | ASCII | UNICODE]
VARCHAR (<length>) [BINARY]
```

The CHAR data type is a fixed-length character data type that can store up to 255 characters. The `<length>` placeholder specifies the number of characters stored. Although the actual value can be made up of fewer characters than the amount specified, the actual storage space is fixed at the specified amount. Take a look at an example to demonstrate how this works. The following table definition includes a column definition for the Category column:

```
CREATE TABLE Catalog
(
    ProductID SMALLINT,
    Description VARCHAR(40),
    Category CHAR(3),
    Price DECIMAL(7,2)
);
```

The Category column is defined with the CHAR(3) data type. As a result, the column can store zero to three characters per value, but the storage amount allotted to that value is always three bytes, one for each character.

The CHAR data type is an appropriate data type to use when you know how many characters most values in a column will consist of and when the values are made up of alphanumeric characters, as opposed to all numerals. If you don't know how many characters each value will be, you should use a VARCHAR data type. The VARCHAR data type also allows you to specify a maximum length; however, storage requirements are based on the actual number of characters, rather than on the `<length>` value.

Return to the preceding example. Notice that the Description column is configured with the VARCHAR(40) data type. This means that the values can be of varying length, up to 40 characters long. As a result, the amount of actual storage ranges between zero bytes and 40 bytes, depending on the actual value. For example, a value of "Bookcase" requires fewer bytes than a value of "Three-piece living room set."

The VARCHAR data type, like the CHAR data type, can store up to 255 characters. Along with the flexibility offered by VARCHAR, compared to CHAR, comes a performance cost. CHAR columns are processed more efficiently than VARCHAR columns, yet CHAR columns can result in wasted storage. Generally, for columns with values that vary widely in length, the VARCHAR data type might often be your best choice.

If you return again to the `<character data type>` syntax, you'll see that the CHAR data type allows you to specify the BINARY, ASCII, or UNICODE attribute, and the VARCHAR data type allows you to specify the BINARY attribute. The three attributes result in the following effects:

❑ **BINARY:** Makes sorting and comparisons case sensitive.

❑ **ASCII:** Assigns the `latin1` character set to the column.

❑ **UNICODE:** Assigns the `ucs2` character set to the column.

149

In addition to string data types, MySQL supports four types of binary data types, as shown in the following syntax:

```
<binary data type>::=
  TINYBLOB | BLOB | MEDIUMBLOB | LONGBLOB
```

Binary data types support the storage of large amounts of data, such as image and sound files. These types are useful when you expect values to grow large or to vary widely. The four binary data types are identical except for the maximum amount of data that each one supports. The following table shows the maximum size of values permitted in a column configured with one of these data types.

Data type	Maximum size
TINYBLOB/TINYTEXT	255 characters (355 bytes)
BLOB/TEXT	65,535 characters (64 KB)
MEDIUMBLOB/MEDIUMTEXT	16,777,215 characters (16 MB)
LONGBLOB/LONGTEXT	4,294,967,295 characters (4 GB)

The text data types are also included in the table because they are similar to the binary data types and because the maximum size limitations are the same. The text data types are discussed later in this section.

The binary data types do not take any arguments. As with data in a VARCHAR column, the storage used for binary data varies according to the size of the value, but you do not specify a maximum length. When defining a column with a binary data type, you simply type the name of the data type in the column definition. For example, the following table definition includes a BLOB column named Photo:

```
CREATE TABLE Inventory
(
    ProductID SMALLINT UNSIGNED,
    Name VARCHAR(40),
    Photo BLOB,
    Quantity INT UNSIGNED
);
```

The Photo column can store binary data up to 64 KB in size. The assumption in this case is that a photo can be taken of the product and saved in a small enough file to fit into this column. If you anticipate that the photos might be larger, you should step this up to MEDIUMBLOB.

The text data types are very similar to the binary data types and, as the following syntax shows, have a direct counterpart to each of the four binary data types:

```
<text data type>::=
  TINYTEXT | TEXT | MEDIUMTEXT | LONGTEXT
```

The text data types also have the same size limitations and storage requirements as the binary data types. If you refer to the previous table, you can see how the sizes correspond between the binary data types and the text data types. The main difference between the two types is that the text data types are associated with a specific character set. Binary columns are treated as strings, and sorting is

case sensitive. Text columns are treated according to their character sets, and sorting is based on the collation for that character set.

The following CREATE TABLE statement provides an example of a TEXT column named DescriptionDoc:

```
CREATE TABLE Catalog
(
   ProductID SMALLINT UNSIGNED,
   Name VARCHAR(40),
   DescriptionDoc TEXT CHARACTER SET latin1 COLLATE latin1_bin
);
```

As you can see, the DescriptionDoc column includes a CHARACTER SET and COLLATE clause. The latin1 character set and the latin1_bin collation are specific to the values in the DescriptionDoc column. The advantage of this is that you can use a character set and collation that differ from that of the table, database, or server.

Now take a look at the list data types, which are the last set of string data types. As the following syntax shows, the list data types include ENUM and SET:

```
<list data type>::=
{ENUM | SET} (<value> [{, <value>}...])
```

The ENUM data type allows you to specify a list of values that you can use in a column configured with that type. When you insert a row in the table, you can also insert one of the values defined for the data type in the column. The column can contain only one value, and it must be one of the listed values. A SET data type also specifies a list of values to be inserted in the column. Unlike the ENUM data type, in which you can specify only one value, the SET data type allows you to specify multiple values from the list.

The following table definition illustrates how you can configure an ENUM column and a SET column:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED,
   BikeModel VARCHAR(40),
   BikeColor ENUM('red', 'blue', 'green', 'yellow'),
   BikeOptions SET('rack', 'light', 'helmet', 'lock')
);
```

Notice that the list of values follows the data type. The values are enclosed in single quotes and separated by commas, and all values are enclosed in parentheses. For an ENUM data type, you can specify up to 65,535 values. For a SET data type, you can specify up to 64 values.

Date/Time Data Types

The final category of data types is the date/time data types, which are shown in the following syntax:

```
<date/time data type>::=
DATE | TIME | DATETIME | YEAR | TIMESTAMP
```

The date/time data types allow you to specify columns that contain data specific to dates and times. The date/time data types support the ranges and formats shown in the following table:

Data type	Format	Range
DATE	YYYY-MM-DD	1000-01-01 through 9999
TIME	HH:MM:SS	–838:59:59 to 838:59:59
DATETIME	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 through 9999
YEAR	YYYY	1901 to 2155 (and 0000)
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1970-01-01 00:00:00 to partway through 2037

The date/time data types are handy if you want to store specific types of date information. For example, if you want to record only the date and not the year, you would use the DATE data type. The values that you entered in this column would have to conform to the format defined by that data type. However, of particular interest is the TIMESTAMP data type, which is slightly different from the other data types. When you configure a column with this data type, a row, when inserted or updated, is automatically provided a value for the TIMESTAMP column that is based on the current time and date. This provides a handy way to record each transaction that occurs in a particular table.

Now take a look at a table that uses time/date data types. The following table definition includes a YEAR column and a TIMESTAMP:

```
CREATE TABLE BookOrders
(
    OrderID SMALLINT UNSIGNED,
    BookID SMALLINT UNSIGNED,
    Copyright YEAR,
    OrderDate TIMESTAMP
);
```

The Copyright column allows you to add a value to the column that falls in the range of 1901 to 2155; however, you're restricted from adding any other types of values. The OrderDate column automatically records the current data and time when a particular row is inserted or updated, so you don't have to insert any values in this column.

Defining a Column's Nullability

Up to this point, the focus has been on identifying only the required elements of a CREATE TABLE statement and the column definitions. As a result, the column definitions have included only the column names and the data types assigned to those columns. The next component of the column definition examined in this chapter is the column's nullability, which is specified through the NULL and NOT NULL keywords.

A column's nullability refers to a column's ability to accept null values. Recall from Chapter 1 that a null value indicates that a value is undefined or unknown. It is not the same as zero or blank, but instead means that the value is absent. When you include NOT NULL in your column definition, you're saying that the column does not permit null values. In other words, a specific value must always be provided for that column. On the other hand, the NULL option permits null values. If neither option is specified, NULL is assumed, and null values are permitted in the column.

Now take a look at a table definition that specifies the nullability of its columns. The following example creates the Catalog table and includes two NOT NULL columns:

```
CREATE TABLE Catalog
(
    ProductID SMALLINT UNSIGNED NOT NULL,
    Name VARCHAR(40) NOT NULL
);
```

You must provide a value for both the ProductID column and the Name column. Whenever you insert rows in the table or update rows in this table, you cannot use NULL as a value for either of those columns. In general, whenever you configure a column as NOT NULL, you must supply a value other than NULL to the column when inserting and modifying rows. There are, however, two exceptions to this rule. If you configure a column with the TIMESTAMP data type or if you use the AUTO_INCREMENT option, then inserting NULL automatically sets the value of the column to the correct TIMESTAMP value or the AUTO_INCREMENT value. (The AUTO_INCREMENT option is discussed later in the chapter.) But other than these two exceptions, a NOT NULL column cannot accept a null value.

Defining Default Values

Situations might arise in which you want a column to use a default value when inserting or updating a row, if no value is provided for a column. This is useful when a value is often repeated in a column or it is the value most likely to be used in that column. MySQL allows you to assign default values through the use of a DEFAULT clause. For example, the following table definition includes a column defined with a default value of Unknown:

```
CREATE TABLE AuthorBios
(
    AuthID SMALLINT UNSIGNED NOT NULL,
    YearBorn YEAR NOT NULL,
    CityBorn VARCHAR(40) NOT NULL DEFAULT 'Unknown'
);
```

In this CREATE TABLE statement, the CityBorn column is configured with the VARCHAR data type and the NOT NULL option. In addition, the column definition includes a DEFAULT clause. In that clause, the keyword DEFAULT is specified, followed by the actual default value, which in this case is Unknown. If you insert a row in the table and do not specify a value for the CityBorn column, the value Unknown is automatically inserted in that column.

You can also specify a default value in a column configured with a numeric data type. In the following table definition, the NumBooks column is configured with the SMALLINT data type and a default value of 1:

```
CREATE TABLE AuthorBios
(
    AuthID SMALLINT UNSIGNED NOT NULL,
    YearBorn YEAR NOT NULL,
    NumBooks SMALLINT NOT NULL DEFAULT 1
);
```

Notice that you do not need to enclose the default value in single quotes. The quote marks are used only for defaults that are string values.

153

You can also specify NULL as a default value. The column, though, must permit null values in order to specify it as a default.

If you do not assign a default value to a column, MySQL automatically assigns a default to the column. If a column accepts null values, the default is NULL. If a column does not accept null values, the default depends on how the column is defined:

❑ For columns configured with the TIMESTAMP data type, the default value for the first TIMESTAMP column is the current date and time. The default values for any other TIMESTAMP columns in the table are zero values in place of the date and time.

❑ For columns configured with a date/time data type other than TIMESTAMP, the default values are zero values in place of the date and time.

❑ For columns configured with the AUTO_INCREMENT option, the default value is the next number in the incremented sequence of numbers. (The AUTO_INCREMENT option is discussed later in the chapter.)

❑ For numeric columns that are not configured with the AUTO_INCREMENT option, the default value is 0.

❑ For columns configured with the ENUM data type, the default value is the first value specified in the column definition.

❑ For columns configured with a string data type other than the ENUM type, the default value is an empty string.

As you can see, the rules for defaults in NOT NULL columns are more complex than for columns that permit null values. As a result, you might consider defining defaults on any columns whose default value you want to control.

Most relational database management systems (RDBMSs) do not automatically assign default values to all columns. For these systems, trying to insert a value in a column for which you defined no default and null values are not permitted results in an error. As you can see with MySQL, all columns are assigned a default value.

Defining Primary Keys

In Chapter 4, when you were learning how to create a data model, you were introduced to the concept of primary keys and how they ensure the uniqueness of each row in a table. A primary key is one or more columns in a table that uniquely identify each row in that table. For nearly any table you create, you should define a primary key for that table.

The easiest way to define a single-column primary key is to specify the PRIMARY KEY option in the column definition, as shown in the following example:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ModelID SMALLINT UNSIGNED NOT NULL,
    ModelDescrip VARCHAR(40)
);
```

In this table definition, the primary key for the Orders table is defined on the OrderID column. You only need to add the `PRIMARY KEY` clause to the column definition. In order to define a column as a primary key, the column must be configured as `NOT NULL`. If you do not explicitly specify the `NOT NULL` option, `NOT NULL` is assumed. In addition, a table can have only one primary key, which you can define in the column definition or as a separate constraint, as shown in the following syntax:

```
[CONSTRAINT <constraint name>] PRIMARY KEY (<column name> [{, <column name>}...])
```

When you define a primary key as a separate constraint, you're including it as a table element in your table definition, as you would other table elements, such as column definitions. For example, you can define the same primary key that is shown in the preceding example as a table element. The following table definition for the Orders table removes the `PRIMARY KEY` clause from the OrderID column definition and uses a `PRIMARY KEY` constraint:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL,
    ModelID SMALLINT UNSIGNED NOT NULL,
    ModelDescrip VARCHAR(40),
    PRIMARY KEY (OrderID)
);
```

As you can see, the `PRIMARY KEY` constraint is added as another table element, just like the three columns. The table element needs to include only the keywords `PRIMARY KEY` and the name of the primary key column, enclosed in parentheses. If you were creating a primary key on more than one column, you would include both of those column names in the parentheses, as shown in the following table definition:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL,
    ModelID SMALLINT UNSIGNED NOT NULL,
    ModelDescrip VARCHAR(40),
    PRIMARY KEY (OrderID, ModelID)
);
```

Notice that the `PRIMARY KEY` constraint now specifies the OrderID column and the ModelID column. As a result, the primary key for the Orders table will be created on these two columns, which means that no two value pairs can be alike, although values can be repeated in individual columns. Any time that you plan to define a primary key on two or more columns, you must use the table-level constraint. You cannot define a primary key on multiple columns at the column level, as you can when you include only one column in the primary key.

Defining Auto-Increment Columns

In some cases, you may want to generate the numbers in your primary key automatically. For example, each time you add an order to a new table, you want to assign a new number to identify that order. The more rows that the table contains, the more order numbers there will be. For this reason, MySQL allows you to define a primary key column with the `AUTO_INCREMENT` option. The `AUTO_INCREMENT` option allows you to specify that numbers be generated automatically for your foreign key column. For example, the primary key column in the following example is configured with the `AUTO_INCREMENT` option:

```
    CREATE TABLE Catalog
    (
        ProductID SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
        Name VARCHAR(40) NOT NULL,
        PRIMARY KEY (ProductID)
    );
```

In this example, the ProductID column is configured with the SMALLINT data type and is configured as the primary key (through the use of a PRIMARY KEY constraint). The column definition also includes the NOT NULL option and the AUTO_INCREMENT option. As a result, whenever you add a new row to the Catalog table, a new number is automatically assigned to the ProductID column. The number is incremented by 1, based on the highest value existing in that column. For example, if a row exists with a ProductID value of 1347, and this is the highest ProductID value in the table, the next row inserted in the table is assigned a ProductID value of 1348.

You can use the AUTO_INCEREMENT option only on a column configured with an integer data type and the NOT NULL option. In addition, the table must be set up as a primary key or with a unique index, and there can be only one AUTO_INCREMENT column per table. (Unique indexes are discussed later in the chapter.) Also, the AUTO_INCREMENET column cannot be defined with a default value.

Defining Foreign Keys

In Chapter 4, you learned how tables in a relational database form relationships with each other in order to associate data in a meaningful way and to help ensure data integrity. When you created your data model, you showed these relationships by connecting related tables with lines that indicated the type of relationship. In your final data model, you found that the most common type of relationship was the one-to-many, which was represented by a line that had three prongs on the *many* side of the relationship.

In order to implement these relationships in MySQL, you must define foreign keys on the referencing tables. You define the foreign key on the column or columns in the table that references the column or columns in the referenced table. The referencing table, the table that contains the foreign key, is often referred to as the *child* table, and the referenced table is often referred to as the *parent* table.

The foreign key maintains the consistency of the data between the child table the parent table. In order to insert a row in the child table or to update a row in that table, the value in the foreign key column must exist in the referenced column in the parent table.

For example, suppose you have a table that tracks sales for a bookstore. One of the columns in the table stores the IDs for the books that have sold. Data about the books themselves is actually stored in a separate table, and each book is identified in that table by its ID. As a result, the book ID column in the sales table references the book ID column in the books table. To associate the data in these two columns, the book ID column in the sales table is configured as a foreign key. Because of the foreign key, no book ID can be added to the sales table that doesn't exist in the books table. The sales table, then, is the child table, and the books table is the parent table.

To add a foreign key to a table, you can define it in the column definition or as a constraint in a separate table element. To add the foreign key to a column definition, you must add a reference definition, which is shown in the following syntax:

```
<reference definition>::=
REFERENCES <table name> [(<column name> [{, <column name>}...])]
[ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
[ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT }]
```

As you can see from the syntax, the clause includes several required elements: the REFERENCES keyword, the name of the referenced (parent) table, and at least one column in that table, enclosed in parentheses. The syntax also includes an optional ON DELETE clause and an optional ON UPDATE clause. The ON DELETE clause specifies how MySQL treats related data in the child table when a row in the parent table is deleted. The ON UPDATE clause specifies how MySQL treats related data in the child table when a row in the parent table is updated. For each clause, five options are available. You can specify only one option for each clause. These options are described in the following table:

Option	Description
RESTRICT	If the child table contains values in the referencing columns that match values in the referenced columns in the parent table, rows in the parent table cannot be deleted, and values in the referenced columns cannot be updated. This is the default option if an ON DELETE or ON UPDATE clause is not specified.
CASCADE	Rows in the child table that contain values that also exist in the referenced columns of the parent table are deleted when the associated rows are deleted from the parent table. Rows in the child table that contain values that also exist in the referenced columns of the parent table are updated when the associated values are updated in the parent table.
SET NULL	Values in the referencing columns of the child table are set to NULL when rows with referenced data in the parent table are deleted from the parent table or when the referenced data in the parent table is updated. To use this option, all referencing columns in the child table must permit null values.
NO ACTION	No action is taken in the child table when rows are deleted from the parent table or values in the referenced columns in the parent table are updated.
SET DEFAULT	Values in the referencing columns of the child table are set to their default values when rows are deleted from the parent table or the referenced columns of the parent table are updated.

When you define a foreign key, you can include an ON DELETE clause, an ON UPDATE clause, or both. If you include both, you can configure them with the same option or with different options. If you exclude one or both, the RESTRICT option is assumed in either case, which means that updates and deletes are limited to rows with nonreferenced values. In addition, when defining a foreign key, the referencing columns must have data types compatible with the referenced columns. For integer data types, the size and signed/unsigned status must be the same. The length of a string data type, however, doesn't have to be the same. It's generally a good idea to configure the referencing and referenced columns with the same data type and type-related options.

Now that you have an overview of how to use a reference definition to add a foreign key to a column definition, take a look at an example to help demonstrate how this works. In the following CREATE TABLE statement, a reference definition has been added to the ModelID column:

157

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ModelID SMALLINT UNSIGNED NOT NULL REFERENCES Models (ModelID),
    ModelDescrip VARCHAR(40)
);
```

In this example, the ModelID column is configured with a SMALLINT data type (unsigned), a NOT NULL option, and a REFERENCES clause, which specifies the name of the parent table (Models) and the name of the referenced column (ModelID) in the parent table. As a result, the ModelID column of the Orders table can include only values that are listed in the ModelID column of the Models table.

You can also define this foreign key as a separate table element by adding a FOREIGN KEY constraint to your table definition. The following syntax shows how to define a FOREIGN KEY constraint:

```
[CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
    (<column name> [{, <column name>}...]) <reference definition>
```

As the syntax indicates, you must include the keywords FOREIGN KEY, the name of the referencing columns in the child table, enclosed in parentheses, and a reference definition. The reference definition is the same definition used in a column definition to add a foreign key. To illustrate this, rewrite the last example table definition, but this time use a FOREIGN KEY constraint to define the foreign key:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ModelID SMALLINT UNSIGNED NOT NULL,
    ModelDescrip VARCHAR(40),
    FOREIGN KEY (ModelID) REFERENCES Models (ModelID)
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

In this example, the FOREIGN KEY constraint is added as a table element, along with the column definitions. The same column (ModelID) is being configured as a foreign key that references the ModelID column of the Models table. The only difference between this example and the last example is that the reference definition in the last example includes an ON DELETE clause and an ON UPDATE clause, both of which are configured with the CASCADE option. As a result, the child table reflects changes to the parent table.

If you want to define a foreign key on more than one column, you must use a FOREIGN KEY constraint, rather than adding a referencing definition to the column definition. In addition, you must separate the column names by commas and enclose all the column names in parentheses.

Defining Table Types

When you were first introduced to the table definition syntax earlier in the chapter, one of the last elements in that syntax was the <table option> placeholder. For each table definition, you can include one or more table options.

For the most part, these options are beyond the scope of this book. If you want to learn more about them, you are encouraged to review the MySQL product documentation.

One of the table options that is especially important when learning about MySQL is the one that allows you to define the type of table that you create in your table definition. Recall from Chapter 3 that MySQL allows you to create six different types of tables, which are shown in the following syntax:

```
ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MYISAM}
```

To define a table type, you must include an `ENGINE` clause at the end of your table definition, after the parentheses that enclose your table elements. For example, the following table definition specifies the `InnoDB` table type:

```
CREATE TABLE AuthorBios
(
    AuthID SMALLINT UNSIGNED NOT NULL,
    YearBorn YEAR NOT NULL,
    CityBorn VARCHAR(40) NOT NULL DEFAULT 'Unknown'
)
ENGINE=INNODB;
```

In this definition, an `ENGINE` clause is added after the last column definition and closing parentheses. Notice that you simply specify the `ENGINE` keyword, the equal sign, and one of the seven table types. Each table type in MySQL supports a specific set of functionality and serves specific purposes. In addition, each type is associated with a related storage engine (handler) that processes the data in that table. For example, the `MyISAM` engine processes data in `MyISAM` tables. The following table discusses each of the six types of tables.

Table type	Description
BDB	A transaction-safe table that is managed by the Berkeley DB (BDB) handler. The BDB handler also supports automatic recovery and page-level locking. The BDB handler does not work on all the operating systems on which MySQL can operate. For the most part, InnoDB tables have replaced BDB tables.
MEMORY	A table whose contents are stored in memory. The data stored in the tables is available only as long as the MySQL server is available. If the server crashes or is shut down, the data disappears. Because these types of tables are stored in memory, they are very fast and are good candidates for temporary tables. MEMORY tables can also be referred to as HEAP tables, although MEMORY is now the preferable keyword.
InnoDB	A transaction-safe table that is managed by the InnoDB handler. As a result, data is not stored in a .MYD file, but instead is managed in the InnoDB tablespace. InnoDB tables also support full foreign key functionality in MySQL, unlike other tables. In addition, the InnoDB handler supports automatic recovery and row-level locking. InnoDB tables do not perform as well as MyISAM tables.
ISAM	A deprecated table type that was once the default table type in MySQL. The MyISAM table type has replaced it, although it is still supported for backward compatibility. Eventually, ISAM tables will no longer be supported.

MERGE	A virtual table that is made up of identical MyISAM tables. Data is not stored in the MERGE table, but in the underlying MyISAM tables. Changes made to the MERGE table definition do not affect the underlying MyISAM tables. MERGE tables can also be referred to as MRG_MyISAM tables
MyISAM	The default table type in MySQL. MyISAM tables, which are based on and have replaced ISAM tables, support extensive indexing and are optimized for compression and speed. Unlike other table types, BLOB and TEXT columns can be indexed and null values are allowed in indexed columns. MyISAM tables are not transaction safe, and they do not support full foreign key functionality.

You can use the TYPE keyword to specify the table type, rather than the ENGINE keyword. However, TYPE has been deprecated in MySQL, which means that it will eventually be phased out. If you use TYPE, you receive a warning about its deprecated state, but the table is still created.

Creating Tables in the DVDRentals Database

Now that you have learned how to create a table in MySQL, it's time to try it out for yourself. The following three Try It Out sections walk you through the steps necessary to create the tables in the DVDRentals database. The tables are based on the final database design that you developed in Chapter 4. The tables are divided into three categories that correspond to the following Try It Out sections. The first group of tables acts as lookup tables in the database. They must be developed before you create tables that reference the lookup table. The second category of tables holds data about the people who will participate somehow in the database system. These include the movie participants, the employees, and the customers. The last group includes the tables that contain foreign keys. You must create this group of tables last because they contain columns that reference other tables. In addition, you must create these tables in a specific order because of how they reference each other.

In this Try It Out, you create the six lookup tables that are part of the DVDRentals database, which you created in the Try It Out section earlier in the chapter. These tables include the Roles, MovieTypes, Studios, Ratings, Formats, and Status tables. As you work your way through this exercise, you should reference the data model that you created in Chapter 4. From there, you can compare the SQL statement that you use here to that model.

Try It Out   Creating the Lookup Tables

Follow these steps to create the six lookup tables:

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. To create the Roles table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Roles
   (
   RoleID VARCHAR(4) NOT NULL,
   RoleDescrip VARCHAR(30) NOT NULL,
   PRIMARY KEY (RoleID)
   )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

3. To create the MovieTypes table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE MovieTypes
    (
    MTypeID VARCHAR(4) NOT NULL,
    MTypeDescrip VARCHAR(30) NOT NULL,
    PRIMARY KEY (MTypeID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

4. To create the Studios table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Studios
    (
    StudID VARCHAR(4) NOT NULL,
    StudDescrip VARCHAR(40) NOT NULL,
    PRIMARY KEY (StudID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

5. To create the Ratings table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Ratings
    (
    RatingID VARCHAR(4) NOT NULL,
    RatingDescrip VARCHAR(30) NOT NULL,
    PRIMARY KEY (RatingID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

6. To create the Formats table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Formats
    (
    FormID CHAR(2) NOT NULL,
    FormDescrip VARCHAR(15) NOT NULL,
    PRIMARY KEY (FormID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

7. To create the Status table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

161

```
    CREATE TABLE Status
        (
        StatID CHAR(3) NOT NULL,
        StatDescrip VARCHAR(20) NOT NULL,
        PRIMARY KEY (StatID)
        )
    ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

How It Works

In this exercise, you created the six lookup tables in the DVDRentals database. The table definitions should be consistent with the final data model that you created in Chapter 4 for the DVDRentals database. In addition, the six tables are very similar. Take a look at one of them, and review the code that you used to create the table. You used the following CREATE TABLE statement to create the Roles table:

```
    CREATE TABLE Roles
        (
        RoleID VARCHAR(4) NOT NULL,
        RoleDescrip VARCHAR(30) NOT NULL,
        PRIMARY KEY (RoleID)
        )
    ENGINE=INNODB;
```

The statement begins with the CREATE TABLE statement, which identifies the name of the new table (Roles). The table definition then includes three table elements, which are separated by commas and enclosed in parentheses. The first two table elements are column definitions. The RoleID column definition creates a column that is configured with a VARCHAR data type. The data type permits up to four characters. In addition, the column does not permit null values. You use a VARCHAR data type for the RoleID column, rather than a CHAR data type because MySQL converts CHAR data types to VARCHAR data types whenever more than three characters are specified for the value length *and* there are other varying-length columns in the table (which is the case for the RoleDescrip column). Otherwise, you would use CHAR(4) because the values in the column have a fixed length of four characters.

The second column defined in the Roles table definition is the RoleDescrip column, which is configured with a VARCHAR data type and a maximum length of 30 characters. This column also does not permit null values. The last table element in the CREATE TABLE statement is the PRIMARY KEY constraint, which defines a primary key on the RoleID column. As a result, this column uniquely identifies each role in the table.

The last component of the Roles table definition is the ENGINE table option, which species that the table type is InnoDB. You specify this table type because InnoDB is the only type that supports transactions and foreign keys, both of which are important to the DVDRentals database.

The other five tables that you created in the exercise are nearly identical to the Roles table, except for the names of the tables and columns. The only other difference is that string columns with a length less than four are configured with CHAR data types rather than VARCHAR.

Once you create the six lookup tables, you can create the three tables that contain the people (explained in the following Try It Out). Because these tables do not contain foreign keys that reference other tables, you could have created these three tables first. In fact, you could have created the nine tables in any order, as long as all referenced (parent) tables are created before the referencing (child) tables. You grouped the tables together in the manner you did just to keep similar types of tables together in order to make explanations simpler.

Try It Out   **Creating the People Tables**

The following steps describe how to create the three tables that contain people:

1. To create the Participants table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Participants
    (
    PartID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    PartFN VARCHAR(20) NOT NULL,
    PartMN VARCHAR(20) NULL,
    PartLN VARCHAR(20) NULL
    )
ENGINE=INNODB;
```

   You should receive a message indicating that the statement executed successfully.

2. To create the Employees table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Employees
    (
    EmpID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    EmpFN VARCHAR(20) NOT NULL,
    EmpMN VARCHAR(20) NULL,
    EmpLN VARCHAR(20) NOT NULL
    )
ENGINE=INNODB;
```

   You should receive a message indicating that the statement executed successfully.

3. To create the Customers table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Customers
    (
    CustID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    CustFN VARCHAR(20) NOT NULL,
    CustMN VARCHAR(20) NULL,
    CustLN VARCHAR(20) NOT NULL
    )
ENGINE=INNODB;
```

   You should receive a message indicating that the statement executed successfully.

163

How It Works

As with the previous exercise, this exercise adds several tables to the DVDRentals database. Except for the names of the tables and columns, the table definitions are nearly identical. As a result, this explanation covers only one of these definitions to understand how the statements work. The following CREATE TABLE statement is the one you used to create the Participants table:

```
CREATE TABLE Participants
    (
    PartID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    PartFN VARCHAR(20) NOT NULL,
    PartMN VARCHAR(20) NULL,
    PartLN VARCHAR(20) NULL
    )
ENGINE=INNODB;
```

The Participants table definition includes four table elements, separated by commas and enclosed in parentheses. All four table elements are column definitions. The first column definition defines the PartID column, which is configured with the SMALLINT data type, the NOT NULL option, and the AUTO_INCREMENT option. The column is also defined as the primary key. As a result, values in the column uniquely identify each row in the table, null values are not allowed, and the values inserted in the column are generated automatically.

The remaining three columns are configured with the VARCHAR data type and are assigned a length of 20 characters. Null values are not allowed in the PartFN column, but they are allowed in the PartMN columns and the PartLN columns. The columns are set up this way to allow for actors and other movie participants who are known by only one name. (Cher comes to mind as one example.) The other two tables—Employees and Customers—are different in this respect because a last name is required. This, of course, is a business decision, and the business rules collected for this project would dictate which names are actually required. In the case of these three tables, a middle name is not required for any of them.

As with the six lookup table that you created in the previous exercise, all three of the tables in this exercise have been created as InnoDB tables. To support foreign key functionality, all tables participating in relationships must be configured as InnoDB tables.

Now that you've created all the referenced tables in the DVDRentals database, you're ready to create the referencing tables, which are each configured with one or more foreign keys. The order in which you create these remaining four tables is important because dependencies exist among these four tables. For example, you must create the DVDs table before you create the DVDParticipant and Transactions tables because both these tables reference the DVDs table. In addition, you must create the Orders table before you create the Transactions table because the Transactions table references the Orders table. The following Try It Out shows you how to create all the necessary foreign key tables.

Try It Out Creating the Foreign Key Tables

The following steps describe how to create the four referencing tables:

1. To create the DVDs table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE DVDs
    (
    DVDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    DVDName VARCHAR(60) NOT NULL,
    NumDisks TINYINT NOT NULL DEFAULT 1,
    YearRlsd YEAR NOT NULL,
    MTypeID VARCHAR(4) NOT NULL,
    StudID VARCHAR(4) NOT NULL,
    RatingID VARCHAR(4) NOT NULL,
    FormID CHAR(2) NOT NULL,
    StatID CHAR(3) NOT NULL,
    FOREIGN KEY (MTypeID) REFERENCES MovieTypes (MTypeID),
    FOREIGN KEY (StudID) REFERENCES Studios (StudID),
    FOREIGN KEY (RatingID) REFERENCES Ratings (RatingID),
    FOREIGN KEY (FormID) REFERENCES Formats (FormID),
    FOREIGN KEY (StatID) REFERENCES Status (StatID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

This table definition is based on MySQL version 4.1 or later. This statement will not work for versions earlier than 4.1.

2. To create the DVDParticipant table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE DVDParticipant
    (
    DVDID SMALLINT NOT NULL,
    PartID SMALLINT NOT NULL,
    RoleID VARCHAR(4) NOT NULL,
    PRIMARY KEY (DVDID, PartID, RoleID),
    FOREIGN KEY (DVDID) REFERENCES DVDs (DVDID),
    FOREIGN KEY (PartID) REFERENCES Participants (PartID),
    FOREIGN KEY (RoleID) REFERENCES Roles (RoleID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

3. To create the Orders table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Orders
    (
    OrderID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    CustID SMALLINT NOT NULL,
    EmpID SMALLINT NOT NULL,
    FOREIGN KEY (CustID) REFERENCES Customers (CustID),
    FOREIGN KEY (EmpID) REFERENCES Employees (EmpID)
    )
ENGINE=INNODB;
```

165

You should receive a message indicating that the statement executed successfully.

4. To create the Transactions table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE Transactions
    (
    TransID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    OrderID INT NOT NULL,
    DVDID SMALLINT NOT NULL,
    DateOut DATE NOT NULL,
    DateDue DATE NOT NULL,
    DateIn DATE NOT NULL,
    FOREIGN KEY (OrderID) REFERENCES Orders (OrderID),
    FOREIGN KEY (DVDID) REFERENCES DVDs (DVDID)
    )
ENGINE=INNODB;
```

You should receive a message indicating that the statement executed successfully.

How It Works

Because you created the DVDs table first, that is the first table reviewed. The following CREATE TABLE statement creates a table definition that includes 14 table elements:

```
CREATE TABLE DVDs
    (
    DVDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    DVDName VARCHAR(60) NOT NULL,
    NumDisks TINYINT NOT NULL DEFAULT 1,
    YearRlsd YEAR NOT NULL,
    MTypeID VARCHAR(4) NOT NULL,
    StudID VARCHAR(4) NOT NULL,
    RatingID VARCHAR(4) NOT NULL,
    FormID CHAR(2) NOT NULL,
    StatID CHAR(3) NOT NULL,
    FOREIGN KEY (MTypeID) REFERENCES MovieTypes (MTypeID),
    FOREIGN KEY (StudID) REFERENCES Studios (StudID),
    FOREIGN KEY (RatingID) REFERENCES Ratings (RatingID),
    FOREIGN KEY (FormID) REFERENCES Formats (FormID),
    FOREIGN KEY (StatID) REFERENCES Status (StatID)
    )
ENGINE=INNODB;
```

As you can see, the DVDs table definition includes nine columns. Each column is configured with a data type appropriate to that column. Any column defined as a foreign key is configured with a data type identical to the referenced column. In addition, every column is configured with the NOT NULL option, which means that null values are not permitted. The first column, DVDID, is defined as the primary key and includes the AUTO_INCREMENT option, so unique values are automatically assigned to that column.

Of particular interest in this table definition is the NumDisks column definition, which includes a DEFAULT clause (with a default value of 1). As a result, whenever a row is inserted in the table, the value

for the NumDisks column is set to 1, unless otherwise specified. This was done because most DVDs come with one disk, although some include more.

The table definition also includes five FOREIGN KEY constraints, one for each referencing column. In each case, the constraint specifies the referencing column, the referenced table, and the referenced column. For example, the first FOREIGN KEY constraint specifies the MTypeID column as the referencing column, the MovieTypes table as the referenced table, and the MTypeID column in the MovieTypes table as the referenced column.

The other three tables that you defined in this exercise include column and foreign key definitions similar to what you've seen in the DVD table definition and the table definitions in the previous two exercises. In addition, all four tables are defined as InnoDB tables to support transactions and foreign key functionality. The DVDParticipant table definition includes an element that you have not seen, so that definition is worth a closer look:

```
CREATE TABLE DVDParticipant
    (
    DVDID SMALLINT NOT NULL,
    PartID SMALLINT NOT NULL,
    RoleID VARCHAR(4) NOT NULL,
    PRIMARY KEY (DVDID, PartID, RoleID),
    FOREIGN KEY (DVDID) REFERENCES DVDs (DVDID),
    FOREIGN KEY (PartID) REFERENCES Participants (PartID),
    FOREIGN KEY (RoleID) REFERENCES Roles (RoleID)
    )
ENGINE=INNODB;
```

In this table definition, a composite primary key is defined on the DVDID, PartID, and RoleID columns, all of which are configured as individual foreign keys. As this table demonstrates, primary keys can consist of multiple columns, and those columns can also be configured as foreign keys. Because the three columns, when taken as a whole, uniquely identify each row in the table, you do not have to create an additional column in order to create a primary key. The table, as it exists here, is complete.

Once you create the four remaining tables in the DVDRentals database, you can begin adding the data necessary to populate the tables. As you learned earlier in this chapter, the data must exist in the referenced columns before you can insert it in the referencing columns. What this means is that the lookup tables and the tables that contain people's names must be populated before the other tables. Chapter 6 provides more detail on how you insert data in your tables, but for now, the focus switches to modifying table definitions.

Modifying Tables

It is not uncommon to find that, after creating a table, you want to modify the table definition. Fortunately, MySQL allows you to change a number of table elements after creating a table. For example, you can add columns, alter existing columns, add PRIMARY KEY and FOREIGN KEY constraints, or remove columns and constraints.

To modify an existing table definition, you must use the ALTER TABLE statement. The following syntax shows how to create an ALTER TABLE statement and the options available to that statement:

```
ALTER TABLE <table name>
<alter option> [{, <alter option>}...]

<alter option>::=
{ADD [COLUMN] <column definition> [FIRST | AFTER <column name>]}
| {ADD [COLUMN] (<table element> [{, <table element>}...])}
| {ADD [CONSTRAINT <constraint name>] PRIMARY KEY
    (<column name> [{, <column name>}...])}
| {ADD [CONSTRAINT <constraint name>] FOREIGN KEY [<index name>]
    (<column name> [{, <column name>}...]) <reference definition>}
| {ADD [CONSTRAINT <constraint name>] UNIQUE [<index name>]
    (<column name> [{, <column name>}...])}
| {ADD INDEX [<index name>] (<column name> [{, <column name>}...])}
| {ADD FULLTEXT [<index name>] (<column name> [{, <column name>}...])}
| {ALTER [COLUMN] <column name> {SET DEFAULT <value> | DROP DEFAULT}}
| {CHANGE [COLUMN] <column name> <column definition> [FIRST | AFTER <column name>]}
| {MODIFY [COLUMN] <column definition> [FIRST | AFTER <column name>]}
| {DROP [COLUMN] <column name>}
| {DROP PRIMARY KEY}
| {DROP INDEX <index name>}
| {DROP FOREIGN KEY <constraint name>}
| {RENAME [TO] <new table name>}
| {ORDER BY <column name> [{, <column name>}...]}
| {<table option> [<table option>...]}
```

The basic elements of the ALTER TABLE statement are the ALTER TABLE keywords, the name of the table that you want to modify, and one or more alter options. If you chose more than one option, you must separate the options with a comma. Each of the alter options maps directly to a table definition option, except that you must also include an action keyword such as ADD, ALTER, or DROP. In addition, several of the alter options include additional elements that help to define the option. Take a look at an example to help illustrate this concept. Suppose that you create the following table:

```
CREATE TABLE Books
(
    BookID SMALLINT NOT NULL,
    BookName VARCHAR(40) NOT NULL,
    PubID SMALLINT NOT NULL DEFAULT 'Unknown'
)
ENGINE=INNODB;
```

As you can see, the table definition creates a table named Books, the table contains three columns, and the PubID column contains a default value of Unknown. Now suppose that you want to modify the table to include a primary key, foreign key, and an additional column. The following ALTER TABLE statement modifies the table accordingly:

```
ALTER TABLE Books
ADD PRIMARY KEY (BookID),
ADD CONSTRAINT fk_1 FOREIGN KEY (PubID) REFERENCES Publishers (PubID),
ADD COLUMN Format ENUM('paperback', 'hardcover') NOT NULL AFTER BookName;
```

The statement begins with the ALTER TABLE statement, which identifies the name of the table being modified, which in this case is Books. The next line adds a primary key to the table. The primary key is based on the BookID column. The third line in the ALTER TABLE statement adds a FOREIGN KEY constraint to the table. The name of the constraint is fk_1, the foreign key is defined on the PubID column, and the foreign key references the PubID column in the Publishers table.

The final line of the ALTER TABLE statement adds a column to the table. As you can see, a column definition follows the ADD COLUMN keywords. The name of the column is Format. The column is configured with an ENUM data type that is defined with two values: paperback and hardcover. The column is also configured with the NOT NULL option. The AFTER clause, which is unique to the ALTER TABLE statement, specifies that the new column should be added after the column named BookName.

As you can see, the options available to the ALTER TABLE statement are very consistent to their CREATE TABLE statement counterparts, at least in terms of adding and modifying columns. If you plan to remove a component of a table, the options are much simpler, as shown in the following example:

```
ALTER TABLE Books
DROP PRIMARY KEY,
DROP FOREIGN KEY fk_1,
DROP COLUMN Format;
```

In this ALTER TABLE statement, the primary key, the fk_1 FOREIGN KEY constraint, and the Format column are all removed form the table. As these examples demonstrate, the ALTER TABLE syntax contains few elements that you haven't seen, except for the action keywords and the few options specific to the ALTER TABLE statement. In addition, these examples also demonstrate that you can modify most of the components that you define in a CREATE TABLE statement with an ALTER TABLE statement.

Deleting Tables

Deleting a table from the database is simply a matter of executing a DROP TABLE statement. As the following syntax shows, the only elements required in a DROP TABLE statement are the DROP TABLE keywords and the name of the table:

```
DROP [TEMPORARY] TABLE [IF EXISTS] <table name> [{, <table name>}...]
```

The DROP TABLE statement also includes the optional TEMPORARY keyword, which you use if you want to ensure that you drop only a temporary table and do not inadvertently drop a permanent table. The other optional element in the DROP TABLE statement is the IF EXISTS clause. If you specify the clause, you receive a warning message, rather than an error, if you try to drop a table that doesn't exist. The one other aspect of the DROP TABLE syntax to consider is the ability to add optional table names. You can use this statement to drop multiple tables, as long as you separate them by a comma.

Now that you've seen the syntax, look at an example of a DROP TABLE statement to demonstrate how one works. The following example removes a table named Books from your database:

```
DROP TABLE IF EXISTS Books;
```

As you can see, the statement includes the `DROP TABLE` keywords, along with the name of the table. In addition, the statement includes the `IF EXISTS` clause, which means that, if the table doesn't exist, you receive a warning rather than an error when trying to drop a table.

You cannot drop a parent table referenced in a foreign key. You must first remove the foreign key in the child table and then drop the parent table.

In the following exercise, you create a table named InStock, alter the table definition, and then delete the table from your database.

Try It Out   Altering and Dropping Tables

To perform the tasks mentioned here, follow these steps:

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

```
use test
```

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

2. To create the InStock table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE InStock
(
    ProductID SMALLINT
);
```

You should receive a message indicating that the statement executed successfully.

3. Next, add a column, modify the ProductID column, and add a primary key. To make these changes, type the following `ALTER TABLE` statement at the mysql command prompt, and then press Enter:

```
ALTER TABLE InStock
ADD COLUMN Quantity SMALLINT UNSIGNED NOT NULL,
MODIFY ProductID SMALLINT UNSIGNED NOT NULL,
ADD PRIMARY KEY (ProductID);
```

You should receive a message indicating that the statement executed successfully.

4. Next, drop the column and the primary key that you added in the previous step. To make these changes, type the following `ALTER TABLE` statement at the mysql command prompt, and then press Enter:

```
ALTER TABLE InStock
DROP COLUMN Quantity,
DROP PRIMARY KEY;
```

You should receive a message indicating that the statement executed successfully.

5. Finally, remove the InStock table from the test database. To remove the table, type the following `DROP TABLE` statement at the mysql command prompt, and then press Enter:

```
DROP TABLE InStock;
```

You should receive a message indicating that the statement executed successfully.

How It Works

In this exercise, you used a CREATE TABLE statement to create the InStock table in the test database. Once you created the table, you used the following ALTER TABLE statement to modify the InStock table definition:

```
ALTER TABLE InStock
ADD COLUMN Quantity SMALLINT UNSIGNED NOT NULL,
MODIFY ProductID SMALLINT UNSIGNED NOT NULL,
ADD PRIMARY KEY (ProductID);
```

The ALTER TABLE statement includes three alter options. The first one adds a column named Quantity to the InStock table. The column is configured with the SMALLINT data type (unsigned) and the NOT NULL option. The next alter option modifies the ProductID column by configuring the SMALLINT data type to be unsigned and by adding the NOT NULL option. The final alter option adds a primary key to the table. The primary key is based on the ProductID column.

The next step used the following statement to again modify the table:

```
ALTER TABLE InStock
DROP COLUMN Quantity,
DROP PRIMARY KEY;
```

This statement removes the Quantity column from the InStock table and then drops the primary key. After altering the table, you used a DROP TABLE statement to remove the InStock table from the database.

Managing Indexes

Earlier in the chapter, when you were introduced to the CREATE TABLE statement, you no doubt noticed that some of the table elements were related to indexes. An *index* is a device that MySQL uses to speed up searches and reduce the time it takes to execute complex queries. An index works under the same principles as an index you would find at the end of a book. The index provides an organized list of pointers to the actual data. As a result, when MySQL is executing a query, it does not have to scan each table in its entirety to locate the correct data, but it can instead scan the index, thus resulting in quicker and more efficient access.

Indexes, however, do have their trade-offs. First, they can affect the performance of operations that involve the modification of data in a table because the index must be updated whenever the table has been updated. In addition, indexes require additional disk space, which, for large tables, can translate to a substantial amount of storage. Despite these drawbacks, indexes play a critical role in data access, and few tables in a MySQL database are not indexed in some way.

Index Types

MySQL supports five types of indexes that can be created on a table. As you have worked your way through this chapter, you have already created two types of indexes: primary keys and foreign keys. Whenever you create a primary key or a foreign key, you are automatically creating an index on the columns specified in those keys. In fact, when you create a FOREIGN KEY constraint, you have the option

to provide a name for the index that is being created. If you don't provide a name, MySQL assigns a name based on the first referencing column. In addition, MySQL assigns the name PRIMARY to all primary key indexes.

> *When you are setting up a foreign key on columns in an* InnoDB *table, the referencing foreign key columns and the referenced columns in the parent table must both be indexed.*

In addition to primary key and foreign key indexes, MySQL also supports unique indexes, regular (non-unique) indexes, and full-text indexes. The following table provides an overview of each of the five types of indexes.

Index type	Description
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. Also, 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	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.

When creating a table definition that includes indexes, you should place the primary key columns first, followed by the unique index columns, and then followed by any nonunique index columns. This process helps to optimize index performance. Later in the book, in Chapter 15, you learn more about how to use indexes to optimize query performance, but for now, take a look at how you actually create indexes on columns in a table.

Creating Indexes

MySQL supports several methods for adding indexes to a table. You can include the indexes in your column definition, you can use an ALTER TABLE statement to add an index to a table, or you can use the CREATE INDEX statement to add an index to a table.

Defining Indexes When Creating Tables

When using the CREATE TABLE statement to create a table, you can include a number of table elements in your statement. For example, you can include column definitions, a PRIMARY KEY constraint, or FOREIGN KEY constraints. In addition, you can define a primary key and foreign keys in your column definitions. Regardless of the method that you use to create a primary key or a foreign key, whenever you create such a key, you're automatically creating an index on the columns participating in a particular key.

It is worth noting here that, in MySQL, the keyword KEY *and the keyword* INDEX *are often used synonymously.*

Because you're already familiar with how to create primary key and foreign key indexes in a CREATE TABLE statement, take a look at creating unique, regular, and full-text indexes.

Creating Unique Indexes

To create a unique index, you should use a UNIQUE constraint, which is one of the table element options included in the CREATE TABLE statement. The following syntax shows you how to create a UNIQUE constraint:

```
[CONSTRAINT <constraint name>] UNIQUE [INDEX] [<index name>]
    (<column name> [{, <column name>}...])
```

When adding a unique index to a table definition, you need to include only the keyword UNIQUE and the name of the indexed column, enclosed in parentheses. If you're creating the index on more than one column, then you must separate the column names with a comma. In addition, you can include the CONSTRAINT keyword along with the name of the constraint, the INDEX keyword, or an index name. If you don't include a constraint name or an index name, MySQL provides names automatically. Whether or not you include these optional elements, the basic index is the same.

A unique index is also considered a constraint because it ensures that each value in a column is unique, in addition to indexing these values.

Now take a look at an example to demonstrate how to include a unique index in the table definition. The following CREATE TABLE statement defines a unique index on the OrderID and ModelID columns:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL,
    ModelID SMALLINT UNSIGNED NOT NULL,
    ModelDescrip VARCHAR(40),
    PRIMARY KEY (OrderID),
    UNIQUE (OrderID, ModelID)
);
```

The CREATE TABLE statement actually creates two indexes: one primary key and one unique. Notice that the OrderID column participates in two indexes and that the unique index is defined on two columns. As a result, the OrderID column can contain only unique values, and the OrderID and ModelID values, when taken together, can include only unique value pairs. The ModelID column, however, can include duplicate values.

Creating Regular (Nonunique) Indexes

There might be times when you want to index a column but you don't want to require that the values in the column be unique. For those situations you can use a regular index. As with unique indexes, you can include a regular index in a table definition by adding it as a table element, as shown in the following syntax:

```
{INDEX | KEY} [<index name>] (<column name> [{, <column name>}...])
```

173

When you define a regular index, you must specify the INDEX or KEY keyword and the name of the indexed column, enclosed in parentheses. If you want to index more than one column, you must separate the columns by a comma. For example, suppose that your database includes a table that lists the first name and the last name of a company's customers. You might want to create a composite index (an index on more than one column) on the column that contains the first name and the column that contains the last name so that the entire name can be easily searched.

In addition to specifying the indexed columns, you can provide a name for your index. If you don't provide a name, MySQL names the index automatically.

The following CREATE TABLE statement demonstrates how to include a regular index in a table definition:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL,
    ModelID SMALLINT UNSIGNED NOT NULL,
    PRIMARY KEY (OrderID),
    INDEX (ModelID)
);
```

This statement creates a regular index on the ModelID column. You do not need to specify any other elements to add the index. MySQL provides a name for the index automatically.

Creating Full-Text Indexes

Now take a look at how to add a full-text index to a MyISAM table. As you recall, you can add this type of index only to the CHAR, VARCHAR, or TEXT columns. The following syntax shows how you add a full-text index to a table definition:

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

Adding a full-text index is almost identical to adding a regular index, except that you have to specify the keyword FULLTEXT. The following CREATE TABLE statement demonstrates how this works:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL,
    ModelID SMALLINT UNSIGNED NOT NULL,
    ModelName VARCHAR(40),
    PRIMARY KEY (OrderID),
    FULLTEXT (ModelName)
);
```

This example defines a full-text index on the ModelName column, which is configured with a VARCHAR data type. In addition, because you specify no table type option in this table definition, MySQL uses the default table type, which is MyISAM.

Now that you have an overview of how to add an index in a CREATE TABLE statement, you can try it out. In this exercise, you create a table named CDs. The table definition includes a regular index on the CDName column.

Try It Out   **Creating a Table with an Index**

Follow these steps to complete this exercise:

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

```
use test
```

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

2. Next create the CDs table, which includes an index on the CDName column. To create the CDs table, type the following CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE CDs
(
    CDID SMALLINT UNSIGNED NOT NULL,
    CDName VARCHAR(40) NOT NULL,
    INDEX (CDName)
);
```

You should receive a message indicating that the statement executed successfully.

3. Next, remove the CDs table from the test database. To remove the table, type the following DROP TABLE statement at the mysql command prompt, and then press Enter:

```
DROP TABLE CDs;
```

You should receive a message indicating that the statement executed successfully.

How It Works

In this exercise, you created the CDs table in the test database. To create this table, you used the following CREATE TABLE statement:

```
CREATE TABLE CDs
(
    CDID SMALLINT UNSIGNED NOT NULL,
    CDName VARCHAR(40) NOT NULL,
    INDEX (CDName)
);
```

The statement created a MyISAM table that includes two columns. The last table element in the table definition defines a regular index on the CDName column. After you created the table, you removed it from the database with a DROP TABLE statement.

This chapter covers only how to add an index to your table definition. In Chapter 15, you learn more about indexing and how indexes can be used to optimize performance.

Adding Indexes to Existing Tables

In addition to including indexes in a table definition, you can add an index to an existing table. You can use two methods to add an index: the ALTER TABLE statement and the CREATE INDEX statement.

Using the ALTER TABLE Statement

The `ALTER TABLE` statement allows you to add primary key, foreign key, unique, regular, and full-text indexes to a table. You already saw examples of this earlier in the chapter when you used the `ALTER TABLE` statement to add primary keys and foreign keys to a table. Now look at another example. Suppose that you used the following table definition to create a table named Orders:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ModelID SMALLINT UNSIGNED NOT NULL
);
```

As you can see, the table includes a primary key on the OrderID column, which means that this column has an index defined on it. Now suppose that you want to add a unique index to the table that is placed on both the OrderID column and the ModelID column. To do this, you would use the following `ALTER TABLE` statement:

```
ALTER TABLE Orders
ADD UNIQUE (OrderID, ModelID);
```

By adding the unique index, values in the two columns, when taken together, must be unique, although the ModelID column can still contain duplicate values.

In the following Try It Out you create a table named CDs, use an `ALTER TABLE` statement to add a full-text index, and then drop the table from the database.

Try It Out   Creating an Index with the ALTER TABLE Statement

To complete these tasks, follow these steps:

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

```
use test
```

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

2. Next create the CDs table. To create the table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE CDs
(
    CDID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    CDName VARCHAR(40) NOT NULL
);
```

You should receive a message indicating that the statement executed successfully.

3. Now you use an `ALTER TABLE` statement to add a `FULLTEXT` index to the CDs table. Type the following `ALTER TABLE` statement at the mysql command prompt, and then press Enter:

```
ALTER TABLE CDs
ADD FULLTEXT (CDName);
```

You should receive a message indicating that the statement executed successfully.

4. Finally, remove the CDs table from the test database. To remove the table, type the following DROP TABLE statement at the mysql command prompt, and then press Enter:

```
DROP TABLE CDs;
```

You should receive a message indicating that the statement executed successfully.

How It Works

After creating the CDs table in the test database, you used the following ALTER TABLE statement to add a full-text index to the table:

```
ALTER TABLE CDs
ADD FULLTEXT (CDName);
```

As you can see from the statement, you placed the full-text index on the CDName column. Because this column is configured with a VARCHAR data type, a full-text index could be supported. In addition, the table is set up as a MyISAM table because it is the default table type and you specified no other table type. MyISAM tables are the only tables that support full-text indexing. After adding the index to the table, you removed the table from the test database.

Using the CREATE INDEX Statement

The CREATE INDEX statement allows you to add unique, regular, and full-text indexes to a table, but not primary key or foreign key indexes. The following syntax shows you how to define a CREATE INDEX statement:

```
CREATE [UNIQUE | FULLTEXT] INDEX <index name>
ON <table name> (<column name> [{, <column name>}...])
```

To create a regular index, you need to include only the CREATE INDEX keyword, a name for the index, and an ON clause that specifies the name of the table and the columns to be indexed. If you want to create a unique index, you must also include the UNIQUE keyword, and if you want to create a full-text index, you must include the FULLTEXT keyword. In either case, you must place UNIQUE or FULLTEXT between the CREATE INDEX keywords, as in CREATE UNIQUE INDEX or CREATE FULLTEXT INDEX.

To demonstrate how to use a CREATE INDEX statement, take a look at an example based on the following table definition:

```
CREATE TABLE Orders
(
   OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
   ModelID SMALLINT UNSIGNED NOT NULL
);
```

The CREATE TABLE statement shown here creates a table named Orders that contains the OrderID column and the ModelID column. The OrderID column is configured with a primary key. To add a regular index to the ModelID column, you can use the following CREATE INDEX statement:

```
CREATE INDEX index_1 ON Orders (ModelID);
```

177

Executing this statement creates a regular index named `index_1` on the ModelID column of the Orders table.

In the following exercise, you create a table named CDs, add a regular index to the table, and then drop the table from the database.

Try It Out   **Creating an Index with the CREATE INDEX Statement**

To complete these tasks, follow these steps:

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

```
use test
```

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

2.   Next create the CDs table. To create the table, type the following `CREATE TABLE` statement at the mysql command prompt, and then press Enter:

```
CREATE TABLE CDs
(
    CDID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    CDName VARCHAR(40) NOT NULL
);
```

You should receive a message indicating that the statement executed successfully.

3.   Now use a `CREATE INDEX` statement to add an index to the CDs table. Type the following `CREATE INDEX` statement at the mysql command prompt, and then press Enter:

```
CREATE INDEX index_1 ON CDs (CDName);
```

You should receive a message indicating that the statement executed successfully.

4.   Finally, remove the CDs table from the test database. To remove the table, type the following `DROP TABLE` statement at the mysql command prompt, and then press Enter:

```
DROP TABLE CDs;
```

You should receive a message indicating that the statement executed successfully.

How It Works

After you switched to the test database, you created a table named CDs. The table includes the CDID and the CDName columns, with a primary key defined on the CDID column. You then used the following `CREATE INDEX` statement to add a regular index to the table:

```
CREATE INDEX index_1 ON CDs (CDName);
```

The statement creates a regular index named `index_1` (specified in the `CREATE INDEX` statement) on the CDName column of the CDs table (specified in the `ON` clause). After adding the index, you dropped the table from the test database, which also dropped the index.

Removing Indexes

MySQL provides a couple of methods for removing an index from a table. The first of these is the ALTER TABLE statement, and the other is the DROP INDEX statement. To demonstrate how each of these statements works, take a look at the following CREATE TABLE statement:

```
CREATE TABLE Orders
(
    OrderID SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ModelID SMALLINT UNSIGNED NOT NULL,
    UNIQUE unique_1 (OrderID, ModelID)
);
```

The statement defines a table named Orders. The table contains two columns: OrderID and ModelID. The table is defined with a primary key on the OrderID column and a unique index on the OrderID and ModelID columns. Now suppose that you want to drop the unique index. You could use the following ALTER TABLE statement:

```
ALTER TABLE Orders
DROP INDEX unique_1;
```

In this statement, a DROP INDEX clause drops the index named unique_1. As you can see, you merely need to specify the DROP INDEX keywords and the name of the index (in addition to the ALTER TABLE statement). You can also use a DROP INDEX statement to remove an index, as shown in the following syntax:

```
DROP INDEX <index name> ON <table name>
```

To use this statement, you must specify the DROP INDEX keywords, the name of the index, and the name of the table in the ON clause, as shown in the following example:

```
DROP INDEX unique_1 ON Orders;
```

This DROP INDEX statement removes the unique_1 index from the Orders table, just as the ALTER TABLE statement does previously.

Retrieving Information About Database Objects

Up till this point in the chapter, you've been creating, modifying, and deleting objects in MySQL. MySQL, however, also provides methods that allow you to view information about those objects. In this section, you learn about a number of statements that display information about databases and their tables. The statements can be divided into two broad categories: SHOW statements and DESCRIBE statements.

Using SHOW Statements

The SHOW statements in MySQL display a variety of information about databases and their tables. The database-related SHOW statements include the SHOW CREATE DATABASE and SHOW DATABASES statement. The table-related SHOW statements include SHOW COLUMNS, SHOW CREATE TABLE, SHOW INDEX, and SHOW TABLES. When using the table-related statements, you should be working in the context of the database, unless you specify the database name as part of the SHOW statement.

Using Database-Related SHOW Statements

The SHOW CREATE DATABASE statement allows you to view the database definition for a specific database. The following syntax shows how to create a SHOW CREATE DATABASE statement:

```
SHOW CREATE DATABASE <database name>
```

As you can see, you need to specify only the SHOW CREATE DATABASE keywords and the name of the database, as shown in the following example:

```
SHOW CREATE DATABASE mysql;
```

In this case, the statement retrieves information about the mysql database definition. When you execute the statement, you should receive results similar to the following:

```
+----------+----------------------------------------------------------------+
| Database | Create Database                                                |
+----------+----------------------------------------------------------------+
| mysql    | CREATE DATABASE `mysql` /*!40100 DEFAULT CHARACTER SET latin1 */ |
+----------+----------------------------------------------------------------+
1 row in set (0.00 sec)
```

The entire results cannot be displayed here, because the row is too long. The actual results that you see depend on your system, but the basic information is the same.

The next statement to examine is the SHOW DATABASE statement. The statement lists the MySQL databases that exist in your system. The following syntax illustrates the SHOW DATABASE statement:

```
SHOW DATABASES [LIKE '<value>']
```

Notice that the statement includes an optional LIKE clause. The clause lets you specify a value for database names. MySQL returns only names of databases that match that value. You usually use the LIKE clause in conjunction with a wildcard to return names that are similar to the specified value. (In SQL, the percent [%] character serves as a wildcard in much the same way as the asterisk [*] character serves as a wildcard in other applications.) For example, the following SHOW DATABASE returns only those databases whose name begins with "my:"

```
SHOW DATABASES LIKE 'my%';
```

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

```
+----------------+
| Database (my%) |
+----------------+
| mysql          |
+----------------+
1 row in set (0.00 sec)
```

As you can see, only databases that begin with "my" are returned. In this case, only the mysql database is displayed.

Using Table-Related SHOW Statements

The SHOW COLUMNS statement lists the columns in a table, along with information about the columns. The following syntax illustrates the SHOW COLUMNS statement:

```
SHOW [FULL] COLUMNS FROM <table name> [FROM <database name>] [LIKE '<value>']
```

To use this statement, you must specify the SHOW COLUMNS FROM keywords, the name of the table, and optionally the name of the database, if you're not working in the context of that database. To show more complete information about each column, you should also use the FULL keyword. In addition, you can use the LIKE clause to limit the values returned, as shown in the following example:

```
SHOW COLUMNS FROM user FROM mysql LIKE '%priv';
```

This SHOW COLUMNS statement returns column information from the user table in the mysql database. The LIKE clause limits the columns returned to those ending with "priv." (The % wildcard indicates that the value can begin with any characters.) The SHOW COLUMNS statement shown here should produce results similar to the following:

```
+----------------------+--------------+------+-----+---------+-------+
| Field                | Type         | Null | Key | Default | Extra |
+----------------------+--------------+------+-----+---------+-------+
| Select_priv          | enum('N','Y') |      |     | N       |       |
| Insert_priv          | enum('N','Y') |      |     | N       |       |
| Update_priv          | enum('N','Y') |      |     | N       |       |
| Delete_priv          | enum('N','Y') |      |     | N       |       |
| Create_priv          | enum('N','Y') |      |     | N       |       |
| Drop_priv            | enum('N','Y') |      |     | N       |       |
| Reload_priv          | enum('N','Y') |      |     | N       |       |
| Shutdown_priv        | enum('N','Y') |      |     | N       |       |
| Process_priv         | enum('N','Y') |      |     | N       |       |
| File_priv            | enum('N','Y') |      |     | N       |       |
| Grant_priv           | enum('N','Y') |      |     | N       |       |
| References_priv      | enum('N','Y') |      |     | N       |       |
| Index_priv           | enum('N','Y') |      |     | N       |       |
| Alter_priv           | enum('N','Y') |      |     | N       |       |
| Show_db_priv         | enum('N','Y') |      |     | N       |       |
| Super_priv           | enum('N','Y') |      |     | N       |       |
| Create_tmp_table_priv | enum('N','Y') |      |     | N       |       |
| Lock_tables_priv     | enum('N','Y') |      |     | N       |       |
| Execute_priv         | enum('N','Y') |      |     | N       |       |
| Repl_slave_priv      | enum('N','Y') |      |     | N       |       |
| Repl_client_priv     | enum('N','Y') |      |     | N       |       |
+----------------------+--------------+------+-----+---------+-------+
21 rows in set (0.00 sec)
```

Notice that each column name ends in "priv." Also notice that the results include details about each column. You can also retrieve information about a table by using a SHOW CREATE TABLE statement, which displays the table definition. The following syntax shows how to create a SHOW CREATE TABLE statement:

```
SHOW CREATE TABLE <table name>
```

In this statement, you need to specify the SHOW CREATE TABLE keywords, along with the name of the table, as shown in the following example:

```
SHOW CREATE TABLE func;
```

This SHOW CREATE TABLE statement returns the table definition for the func table. When you execute this statement, you should receive results similar to the following:

```
+-------+----------------------------------------------------------------------
| Table | Create Table
+-------+----------------------------------------------------------------------
| func  | CREATE TABLE `func` (`name` char(64) character set latin1 collate latin1_
+-------+----------------------------------------------------------------------
1 row in set (0.02 sec)
```

The entire results could not fit on the screen because the row is too long. The data that you see depends on your system.

The next statement displays a list of indexes in a table. The SHOW INDEX statement is shown in the following syntax:

```
SHOW INDEX FROM <table name> [FROM <database name>]
```

The only required elements of this statement are the SHOW INDEX FROM keywords and the name of the table. You can also use the FROM clause to specify the name of the database, which you would do if you're not working in the context of that database. For example, the following SHOW INDEX statement displays information about the indexes in the user table in the mysql database:

```
SHOW INDEX FROM user FROM mysql;
```

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

```
+-------+------------+----------+-------------+-------------+-----------+---------
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinal
+-------+------------+----------+-------------+-------------+-----------+---------
| user  |          0 | PRIMARY  |           1 | Host        | A         |        N
| user  |          0 | PRIMARY  |           2 | User        | A         |
+-------+------------+----------+-------------+-------------+-----------+---------
2 rows in set (0.00 sec)
```

Once again, the entire results are not displayed here because the rows are too long. The exact results that you see on your system vary; however, the basic information should be the same.

The next statement is the SHOW TABLES statement, which displays a list of tables in the current database or a specified database. The syntax for the statement is as follows:

```
SHOW TABLES [FROM <database name>] [LIKE '<value>']
```

As you can see, the only elements that you need to specify are the SHOW TABLES keywords. You can also specify the database in the FROM clause, and you can specify a value in the LIKE clause, as shown in the following example:

```
SHOW TABLES FROM mysql LIKE 'help%';
```

In this SHOW TABLES statement, you display all tables in the mysql database that begin with "help," as shown in following results:

```
+------------------------+
| Tables_in_mysql (help%) |
+------------------------+
| help_category          |
| help_keyword           |
| help_relation          |
| help_topic             |
+------------------------+
4 rows in set (0.00 sec)
```

As you can see, the list includes only tables that begin with "help."

Using DESCRIBE Statements

Another statement useful for viewing information about tables is the DESCRIBE statement. The following syntax describes how to define a DESCRIBE statement:

```
DESCRIBE <table name> [<column name> | '<value>']
```

The only required elements of the DESCRIBE statement are the DESCRIBE keyword and the name of the table. You can also specify a column name or a value used to return columns with names similar to the value, in which case you would use a wildcard. The following example shows a DESCRIBE statement that returns information about all columns in the user table that end with "priv":

```
DESCRIBE user '%priv';
```

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

```
+-----------------------+---------------+------+-----+---------+-------+
| Field                 | Type          | Null | Key | Default | Extra |
+-----------------------+---------------+------+-----+---------+-------+
| Select_priv           | enum('N','Y') |      |     | N       |       |
| Insert_priv           | enum('N','Y') |      |     | N       |       |
| Update_priv           | enum('N','Y') |      |     | N       |       |
| Delete_priv           | enum('N','Y') |      |     | N       |       |
| Create_priv           | enum('N','Y') |      |     | N       |       |
| Drop_priv             | enum('N','Y') |      |     | N       |       |
| Reload_priv           | enum('N','Y') |      |     | N       |       |
| Shutdown_priv         | enum('N','Y') |      |     | N       |       |
| Process_priv          | enum('N','Y') |      |     | N       |       |
| File_priv             | enum('N','Y') |      |     | N       |       |
| Grant_priv            | enum('N','Y') |      |     | N       |       |
```

183

```
| References_priv       | enum('N','Y') |      |     | N        |       |
| Index_priv            | enum('N','Y') |      |     | N        |       |
| Alter_priv            | enum('N','Y') |      |     | N        |       |
| Show_db_priv          | enum('N','Y') |      |     | N        |       |
| Super_priv            | enum('N','Y') |      |     | N        |       |
| Create_tmp_table_priv | enum('N','Y') |      |     | N        |       |
| Lock_tables_priv      | enum('N','Y') |      |     | N        |       |
| Execute_priv          | enum('N','Y') |      |     | N        |       |
| Repl_slave_priv       | enum('N','Y') |      |     | N        |       |
| Repl_client_priv      | enum('N','Y') |      |     | N        |       |
+-----------------------+---------------+------+-----+----------+-------+
21 rows in set (0.00 sec)
```

Notice that only columns that end in "priv" are displayed. The DESCRIBE statement is a handy way to view the information that you need quickly.

In this exercise, you try out some of the SHOW and DESCRIBE statements that you learned about in this section of the chapter.

Try It Out   Displaying Database Information

The following steps lead you through a series of statements:

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

    ```
    SHOW DATABASES;
    ```

    You should receive results similar to the following:

    ```
    +------------+
    | Database   |
    +------------+
    | dvdrentals |
    | mysql      |
    | test       |
    +------------+
    3 rows in set (0.00 sec)
    ```

    At the very least, you should see the two databases that are installed by default — mysql and test — and the DVDRentals database, which you created earlier in the chapter.

2. Next, you view the CREATE DATABASE statement for the DVDRentals database. To view the database definition, type the following SHOW CREATE DATABASE statement at the mysql command prompt, and then press Enter:

    ```
    SHOW CREATE DATABASE DVDRentals;
    ```

    You should receive results similar to the following:

    ```
    +------------+--------------------------------------------------------------------
    | Database   | Create Database
    +------------+--------------------------------------------------------------------
    | dvdrentals | CREATE DATABASE `dvdrentals` /*!40100 DEFAULT CHARACTER SET latin1 *
    +------------+--------------------------------------------------------------------
    1 row in set (0.01 sec)
    ```

Because the row is so long, only a part of the results are displayed here. The amount of data that is displayed on your system and the way that it is displayed vary from system to system. As a result, you might have to scroll to the right or up and down to view all the results.

3. Now you will switch to the DVDRentals database. Type the following command at the mysql command prompt, and press Enter:

```
use DVDRentals
```

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

4. To display a list of the tables in the DVDRentals database, type the following command at the mysql command prompt, and then press Enter:

```
SHOW TABLES;
```

You should see results similar to the following:

```
+----------------------+
| Tables_in_dvdrentals |
+----------------------+
| customers            |
| dvdparticipant       |
| dvds                 |
| employees            |
| formats              |
| movietypes           |
| orders               |
| participants         |
| ratings              |
| roles                |
| status               |
| studios              |
| transactions         |
+----------------------+
13 rows in set (0.00 sec)
```

All the tables that you created earlier in the chapter should be displayed. As you can see, there are 13 tables in all.

5. Next, view the table definition for the Orders table. Type the following SHOW CREATE TABLE statement at the mysql command prompt, and then press Enter:

```
SHOW CREATE TABLE Orders;
```

You should receive results similar to the following:

```
+--------+----------------------------------------------------------------------
| Table  | Create Table
+--------+----------------------------------------------------------------------
| Orders | CREATE TABLE `orders` (`OrderID` int(11) NOT NULL auto_increment, `CustI
+--------+----------------------------------------------------------------------
1 row in set (0.01 sec)
```

As before, only part of the results can be displayed here. The amount of data displayed and the way it will be displayed vary from system to system.

6. Next, display the columns that are in the Transactions table. Type the following command at the mysql command prompt, and then press Enter:

```
SHOW COLUMNS FROM Transactions;
```

You should receive results similar to the following:

```
+---------+------------+------+-----+------------+----------------+
| Field   | Type       | Null | Key | Default    | Extra          |
+---------+------------+------+-----+------------+----------------+
| TransID | int(11)    |      | PRI | NULL       | auto_increment |
| OrderID | int(11)    |      | MUL | 0          |                |
| DVDID   | smallint(6)|      | MUL | 0          |                |
| DateOut | date       |      |     | 0000-00-00 |                |
| DateDue | date       |      |     | 0000-00-00 |                |
| DateIn  | date       |      |     | 0000-00-00 |                |
+---------+------------+------+-----+------------+----------------+
6 rows in set (0.00 sec)
```

Notice that each column is listed, along with the data type, the column's nullability, and additional column settings.

7. Another way to view information about a table is to use a DESCRIBE statement. Type the following command at a mysql command prompt, and then press Enter:

```
DESCRIBE DVDParticipant;
```

You should receive results similar to the following:

```
+--------+------------+------+-----+---------+-------+
| Field  | Type       | Null | Key | Default | Extra |
+--------+------------+------+-----+---------+-------+
| DVDID  | smallint(6)|      | PRI | 0       |       |
| PartID | smallint(6)|      | PRI | 0       |       |
| RoleID | varchar(4) |      | PRI |         |       |
+--------+------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

As with the SHOW COLUMNS statement, you see a list of columns, along with information about each column.

8. The final step is to view the indexes that have been created on a table. Type the following SHOW INDEX statement at the mysql command prompt, and then press Enter:

```
SHOW INDEX FROM DVDs;
```

You should see results similar to the following:

```
+-------+------------+----------+--------------+-------------+-----------+---------
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinal
+-------+------------+----------+--------------+-------------+-----------+---------
| DVDs  |          0 | PRIMARY  |            1 | DVDID       | A         |
| DVDs  |          1 | MTypeID  |            1 | MTypeID     | A         |
| DVDs  |          1 | StudID   |            1 | StudID      | A         |
| DVDs  |          1 | RatingID |            1 | RatingID    | A         |
| DVDs  |          1 | FormID   |            1 | FormID      | A         |
| DVDs  |          1 | StatID   |            1 | StatID      | A         |
+-------+------------+----------+--------------+-------------+-----------+---------
6 rows in set (0.00 sec)
```

Notice that the primary key and foreign keys are listed as indexes. As you learned earlier in the chapter, there are a number of different types of indexes, including primary keys and foreign keys.

How It Works

In this exercise, you executed a number of SHOW statements and one DESCRIBE statement to view information about the databases that exist in your server, to view the DVDRentals database definition, and to view information about different tables in the DVDRentals database. As you have seen, these statements can be very useful when trying to find information about an existing database and the tables in that database.

As you work your way through this book, you might find that you use these statements often. In that case, you should refer to this chapter as necessary to reference these commands.

Summary

In this chapter, you learned how to create, modify, and remove databases, tables, and indexes from your system. The chapter provided the syntax of the various statements necessary to perform these tasks, explained how to use the syntax to create SQL statements, and provided examples that demonstrated how to implement the various statements. In addition, you created the tables necessary to support the DVDRentals database. The tables were based on the data model that you created in Chapter 4. The chapter also explained how to use SHOW and DESCRIBE statements to view information about your database. Specifically, the chapter provided you with the information you need to perform the following tasks:

❑   Create a database definition that specifies the character set and collation for that database.

❑   Modify the character set and collation associated with a database.

❑   Delete a database.

❑   Create a table definition that includes column definitions, primary keys, foreign keys, and indexes.

❑   Alter table definitions, including adding, modifying, and removing columns, primary keys, foreign keys, and indexes.

❑   Remove tables from a database.

❑   Generate SQL statements that retrieve information about the databases and tables in your system.

Once you know how to create a database, add tables to the database, and configure the elements in the tables, you can create the tables that you need to support your databases. In these tables you can insert, modify, and delete data. In the next chapter, you learn how to manage data in your MySQL database. From there, you learn how to retrieve data, import and export data, and manage transactions. You can even create applications that can access the data from within the application languages.

Exercises

The following exercises help you build on the information you learned in this chapter. To view the answers, see Appendix A.

1. You are creating a database named NewDB. The database uses the server's default character set but uses the `latin1_general_ci` collation. What SQL statement should you use to create the database?

2. You are creating a table in a MySQL database. The table is named Bikes and includes two columns: BikeID and BikeName. The BikeID column must uniquely identify each row in the table, and values must be automatically assigned to each row. In addition, the table should never contain more than 200 models of bikes. The BikeName column must include a descriptive name for each model of bike. The names vary in length but should never exceed 40 characters. In addition, the table never participates in a transaction or foreign key relationship. What SQL statement should you use to create the table?

3. You plan to add a unique index to a table that was defined with the following CREATE TABLE statement:

```
CREATE TABLE ModelTrains
(
ModelID SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
ModelName VARCHAR(40) NOT NULL
);
```

The index should be named un_1 and should be configured on the ModelName column. What `ALTER TABLE` statement should you use to add the index?

4. You now want to drop the un_1 unique index from the ModelTrains table. What `ALTER TABLE` statement should you use to remove the unique index?

5. What SQL statement should you use if you want to view a list of tables in the current database?