CHAPTER 2

Creating and Populating a Database

This chapter provides you with the information you need to create your first database and to create the tables and associated data used for the examples in this book. You will also learn about various data types and see how to create tables using them. Because the examples in this book are executed against a MySQL database, this chapter is somewhat skewed toward MySQL's features and syntax, but most concepts are applicable to any server.

Creating a MySQL Database

If you already have a MySQL database server available for your use, you can skip the installation instructions and start with the instructions in Table 2-1. Keep in mind, however, that this book assumes that you are using MySQL version 6.0 or later, so you may want to consider upgrading your server or installing another server if you are using an earlier release.

The following instructions show you the minimum steps required to install a MySQL 6.0 server on a Windows computer:

1. Go to the download page for the MySQL Database Server at *http://dev.mysql.com/downloads*. If you are loading version 6.0, the full URL is *http://dev.mysql.com/downloads/mysql/6.0.html*.
2. Download the Windows Essentials (x86) package, which includes only the commonly used tools.
3. When asked "Do you want to run or save this file?" click Run.
4. The MySQL Server 6.0—Setup Wizard window appears. Click Next.
5. Activate the Typical Install radio button, and click Next.
6. Click Install.
7. A MySQL Enterprise window appears. Click Next twice.

8. When the installation is complete, make sure the box is checked next to "Configure the MySQL Server now," and then click Finish. This launches the Configuration Wizard.
9. When the Configuration Wizard launches, activate the Standard Configuration radio button, and then select both the "Install as Windows Service" and "Include Bin Directory in Windows Path" checkboxes. Click Next.
10. Select the Modify Security Settings checkbox and enter a password for the `root` user (make sure you write down the password, because you will need it shortly!), and click Next.
11. Click Execute.

At this point, if all went well, the MySQL server is installed and running. If not, I suggest you uninstall the server and read the "Troubleshooting a MySQL Installation Under Windows" guide (which you can find at *http://dev.mysql.com/doc/refman/6.0/en/windows-troubleshooting.html*).

If you uninstalled an older version of MySQL before loading version 6.0, you may have some further cleanup to do (I had to clean out some old Registry entries) before you can get the Configuration Wizard to run successfully.

Next, you will need to open a Windows command window, launch the `mysql` tool, and create your database and database user. Table 2-1 describes the necessary steps. In step 5, feel free to choose your own password for the `lrngsql` user rather than "xyz" (but don't forget to write it down!).

Table 2-1. Creating the sample database

Step	Description	Action
1	Open the Run dialog box from the Start menu	Choose Start and then Run
2	Launch a command window	Type **`cmd`** and click OK
3	Log in to MySQL as `root`	`mysql -u root -p`
4	Create a database for the sample data	`create database bank;`
5	Create the `lrngsql` database user with full privileges on the `bank` database	`grant all privileges on bank.* to 'lrngsql'@'localhost' identified by 'xyz';`
6	Exit the `mysql` tool	`quit;`
7	Log in to MySQL as `lrngsql`	`mysql -u lrngsql -p;`
8	Attach to the `bank` database	`use bank;`

You now have a MySQL server, a database, and a database user; the only thing left to do is create the database tables and populate them with sample data. To do so, download the script at *http://examples.oreilly.com/learningsql/* and run it from the `mysql` utility. If you saved the file as *c:\temp\LearningSQLExample.sql*, you would need to do the following:

1. If you have logged out of the `mysql` tool, repeat steps 7 and 8 from Table 2-1.
2. Type **`source c:\temp\LearningSQLExample.sql;`** and press Enter.

You should now have a working database populated with all the data needed for the examples in this book.

Using the mysql Command-Line Tool

Whenever you invoke the `mysql` command-line tool, you can specify the username and database to use, as in the following:

```
mysql -u lrngsql -p bank
```

This will save you from having to type `use bank;` every time you start up the tool. You will be asked for your password, and then the `mysql>` prompt will appear, via which you will be able to issue SQL statements and view the results. For example, if you want to know the current date and time, you could issue the following query:

```
mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2008-02-19 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

The `now()` function is a built-in MySQL function that returns the current date and time. As you can see, the `mysql` command-line tool formats the results of your queries within a rectangle bounded by +, -, and | characters. After the results have been exhausted (in this case, there is only a single row of results), the `mysql` command-line tool shows how many rows were returned and how long the SQL statement took to execute.

About Missing from Clauses

With some database servers, you won't be able to issue a query without a `from` clause that names at least one table. Oracle Database is a commonly used server for which this is true. For cases when you only need to call a function, Oracle provides a table called `dual`, which consists of a single column called `dummy` that contains a single row of data. In order to be compatible with Oracle Database, MySQL also provides a `dual` table. The previous query to determine the current date and time could therefore be written as:

```
mysql> SELECT now()
       FROM dual;
+---------------------+
| now()               |
+---------------------+
| 2005-05-06 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

If you are not using Oracle and have no need to be compatible with Oracle, you can ignore the `dual` table altogether and use just a `select` clause without a `from` clause.

When you are done with the `mysql` command-line tool, simply type **`quit;`** or **`exit;`** to return to the Windows command shell.

MySQL Data Types

In general, all the popular database servers have the capacity to store the same types of data, such as strings, dates, and numbers. Where they typically differ is in the specialty data types, such as XML documents or very large text or binary documents. Since this is an introductory book on SQL, and since 98% of the columns you encounter will be simple data types, this book covers only the character, date, and numeric data types.

Character Data

Character data can be stored as either fixed-length or variable-length strings; the difference is that fixed-length strings are right-padded with spaces and always consume the same number of bytes, and variable-length strings are not right-padded with spaces and don't always consume the same number of bytes. When defining a character column, you must specify the maximum size of any string to be stored in the column. For example, if you want to store strings up to 20 characters in length, you could use either of the following definitions:

```
char(20)    /* fixed-length */
varchar(20) /* variable-length */
```

The maximum length for `char` columns is currently 255 bytes, whereas `varchar` columns can be up to 65,535 bytes. If you need to store longer strings (such as emails, XML

documents, etc.), then you will want to use one of the text types (`mediumtext` and `longtext`), which I cover later in this section. In general, you should use the `char` type when all strings to be stored in the column are of the same length, such as state abbreviations, and the `varchar` type when strings to be stored in the column are of varying lengths. Both `char` and `varchar` are used in a similar fashion in all the major database servers.

Oracle Database is an exception when it comes to the use of `varchar`. Oracle users should use the `varchar2` type when defining variable-length character columns.

Character sets

For languages that use the Latin alphabet, such as English, there is a sufficiently small number of characters such that only a single byte is needed to store each character. Other languages, such as Japanese and Korean, contain large numbers of characters, thus requiring multiple bytes of storage for each character. Such character sets are therefore called *multibyte character sets*.

MySQL can store data using various character sets, both single- and multibyte. To view the supported character sets in your server, you can use the `show` command, as in:

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                    | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |      1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |      1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |      2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |      1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |      2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |      1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |      1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |      1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |      1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
| cp866    | DOS Russian                 | cp866_general_ci    |      1 |
```

```
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |      1 |
| macce    | Mac Central European        | macce_general_ci    |      1 |
| macroman | Mac West European           | macroman_general_ci |      1 |
| cp852    | DOS Central European        | cp852_general_ci    |      1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |      1 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |      1 |
| binary   | Binary pseudo charset       | binary              |      1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |      1 |
| cp932    | SJIS for Windows Japanese   | cp932_japanese_ci   |      2 |
| eucjpms  | UJIS for Windows Japanese   | eucjpms_japanese_ci |      3 |
+----------+-----------------------------+---------------------+--------+
36 rows in set (0.11 sec)
```

If the value in the fourth column, `maxlen`, is greater than 1, then the character set is a multibyte character set.

When I installed the MySQL server, the `latin1` character set was automatically chosen as the default character set. However, you may choose to use a different character set for each character column in your database, and you can even store different character sets within the same table. To choose a character set other than the default when defining a column, simply name one of the supported character sets after the type definition, as in:

```
varchar(20) character set utf8
```

With MySQL, you may also set the default character set for your entire database:

```
create database foreign_sales character set utf8;
```

While this is as much information regarding character sets as I'm willing to discuss in an introductory book, there is a great deal more to the topic of internationalization than what is shown here. If you plan to deal with multiple or unfamiliar character sets, you may want to pick up a book such as Andy Deitsch and David Czarnecki's *Java Internationalization (http://oreilly.com/catalog/9780596000196/)* (O'Reilly) or Richard Gillam's *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard* (Addison-Wesley).

Text data

If you need to store data that might exceed the 64 KB limit for `varchar` columns, you will need to use one of the text types.

Table 2-2 shows the available text types and their maximum sizes.

Table 2-2. MySQL text types

Text type	Maximum number of bytes
`Tinytext`	255
`Text`	65,535
`Mediumtext`	16,777,215
`Longtext`	4,294,967,295

When choosing to use one of the text types, you should be aware of the following:

- If the data being loaded into a text column exceeds the maximum size for that type, the data will be truncated.
- Trailing spaces will not be removed when data is loaded into the column.
- When using `text` columns for sorting or grouping, only the first 1,024 bytes are used, although this limit may be increased if necessary.
- The different text types are unique to MySQL. SQL Server has a single `text` type for large character data, whereas DB2 and Oracle use a data type called `clob`, for Character Large Object.
- Now that MySQL allows up to 65,535 bytes for `varchar` columns (it was limited to 255 bytes in version 4), there isn't any particular need to use the `tinytext` or `text` type.

If you are creating a column for free-form data entry, such as a `notes` column to hold data about customer interactions with your company's customer service department, then `varchar` will probably be adequate. If you are storing documents, however, you should choose either the `mediumtext` or `longtext` type.

Oracle Database allows up to 2,000 bytes for `char` columns and 4,000 bytes for `varchar2` columns. SQL Server can handle up to 8,000 bytes for both `char` and `varchar` data.

Numeric Data

Although it might seem reasonable to have a single numeric data type called "numeric," there are actually several different numeric data types that reflect the various ways in which numbers are used, as illustrated here:

A column indicating whether a customer order has been shipped
: This type of column, referred to as a *Boolean*, would contain a `0` to indicate `false` and a `1` to indicate `true`.

A system-generated primary key for a transaction table
: This data would generally start at `1` and increase in increments of one up to a potentially very large number.

An item number for a customer's electronic shopping basket
: The values for this type of column would be positive whole numbers between 1 and, at most, 200 (for shopaholics).

Positional data for a circuit board drill machine
: High-precision scientific or manufacturing data often requires accuracy to eight decimal points.

To handle these types of data (and more), MySQL has several different numeric data types. The most commonly used numeric types are those used to store whole numbers. When specifying one of these types, you may also specify that the data is *unsigned*, which tells the server that all data stored in the column will be greater than or equal to zero. Table 2-3 shows the five different data types used to store whole-number integers.

Table 2-3. MySQL integer types

Type	Signed range	Unsigned range
`Tinyint`	−128 to 127	0 to 255
`Smallint`	−32,768 to 32,767	0 to 65,535
`Mediumint`	−8,388,608 to 8,388,607	0 to 16,777,215
`Int`	−2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
`Bigint`	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

When you create a column using one of the integer types, MySQL will allocate an appropriate amount of space to store the data, which ranges from one byte for a `tinyint` to eight bytes for a `bigint`. Therefore, you should try to choose a type that will be large enough to hold the biggest number you can envision being stored in the column without needlessly wasting storage space.

For floating-point numbers (such as 3.1415927), you may choose from the numeric types shown in Table 2-4.

Table 2-4. MySQL floating-point types

Type	Numeric range
`Float(`*`p`*`,`*`s`*`)`	−3.402823466E+38 to −1.175494351E-38
	and 1.175494351E-38 to 3.402823466E+38
`Double(`*`p`*`,`*`s`*`)`	−1.7976931348623157E+308 to −2.2250738585072014E-308
	and 2.2250738585072014E-308 to 1.7976931348623157E+308

When using a floating-point type, you can specify a *precision* (the total number of allowable digits both to the left and to the right of the decimal point) and a *scale* (the number of allowable digits to the right of the decimal point), but they are not required. These values are represented in Table 2-4 as *`p`* and *`s`*. If you specify a precision and scale for your floating-point column, remember that the data stored in the column will be

rounded if the number of digits exceeds the scale and/or precision of the column. For example, a column defined as `float(4,2)` will store a total of four digits, two to the left of the decimal and two to the right of the decimal. Therefore, such a column would handle the numbers 27.44 and 8.19 just fine, but the number 17.8675 would be rounded to 17.87, and attempting to store the number 178.375 in your `float(4,2)` column would generate an error.

Like the integer types, floating-point columns can be defined as `unsigned`, but this designation only prevents negative numbers from being stored in the column rather than altering the range of data that may be stored in the column.

Temporal Data

Along with strings and numbers, you will almost certainly be working with information about dates and/or times. This type of data is referred to as *temporal*, and some examples of temporal data in a database include:

- The future date that a particular event is expected to happen, such as shipping a customer's order
- The date that a customer's order was shipped
- The date and time that a user modified a particular row in a table
- An employee's birth date
- The year corresponding to a row in a `yearly_sales` fact table in a data warehouse
- The elapsed time needed to complete a wiring harness on an automobile assembly line

MySQL includes data types to handle all of these situations. Table 2-5 shows the temporal data types supported by MySQL.

Table 2-5. MySQL temporal types

Type	Default format	Allowable values
`Date`	YYYY-MM-DD	`1000-01-01` to `9999-12-31`
`Datetime`	YYYY-MM-DD HH:MI:SS	`1000-01-01 00:00:00` to `9999-12-31 23:59:59`
`Timestamp`	YYYY-MM-DD HH:MI:SS	`1970-01-01 00:00:00` to `2037-12-31 23:59:59`
`Year`	YYYY	`1901` to `2155`
`Time`	HHH:MI:SS	`-838:59:59` to `838:59:59`

While database servers store temporal data in various ways, the purpose of a format string (second column of Table 2-5) is to show how the data will be represented when retrieved, along with how a date string should be constructed when inserting or updating a temporal column. Thus, if you wanted to insert the date March 23, 2005 into a `date` column using the default format `YYYY-MM-DD`, you would use the string

'2005-03-23'. Chapter 7 fully explores how temporal data is constructed and displayed.

Each database server allows a different range of dates for temporal columns. Oracle Database accepts dates ranging from 4712 BC to 9999 AD, while SQL Server only handles dates ranging from 1753 AD to 9999 AD (unless you are using SQL Server 2008's new `datetime2` data type, which allows for dates ranging from 1 AD to 9999 AD). MySQL falls in between Oracle and SQL Server and can store dates from 1000 AD to 9999 AD. Although this might not make any difference for most systems that track current and future events, it is important to keep in mind if you are storing historical dates.

Table 2-6 describes the various components of the date formats shown in Table 2-5.

Table 2-6. Date format components

Component	Definition	Range
YYYY	Year, including century	1000 to 9999
MM	Month	01 (January) to 12 (December)
DD	Day	01 to 31
HH	Hour	00 to 23
HHH	Hours (elapsed)	-838 to 838
MI	Minute	00 to 59
SS	Second	00 to 59

Here's how the various temporal types would be used to implement the examples shown earlier:

- Columns to hold the expected future shipping date of a customer order and an employee's birth date would use the `date` type, since it is unnecessary to know at what time a person was born and unrealistic to schedule a future shipment down to the second.
- A column to hold information about when a customer order was actually shipped would use the `datetime` type, since it is important to track not only the date that the shipment occurred but the time as well.
- A column that tracks when a user last modified a particular row in a table would use the `timestamp` type. The `timestamp` type holds the same information as the `datetime` type (year, month, day, hour, minute, second), but a `timestamp` column will automatically be populated with the current date/time by the MySQL server when a row is added to a table or when a row is later modified.
- A column holding just year data would use the `year` type.

- Columns that hold data regarding the length of time needed to complete a task would use the `time` type. For this type of data, it would be unnecessary and confusing to store a date component, since you are interested only in the number of hours/minutes/seconds needed to complete the task. This information could be derived using two `datetime` columns (one for the task start date/time and the other for the task completion date/time) and subtracting one from the other, but it is simpler to use a single `time` column.

Chapter 7 explores how to work with each of these temporal data types.

Table Creation

Now that you have a firm grasp on what data types may be stored in a MySQL database, it's time to see how to use these types in table definitions. Let's start by defining a table to hold information about a person.

Step 1: Design

A good way to start designing a table is to do a bit of brainstorming to see what kind of information would be helpful to include. Here's what I came up with after thinking for a short time about the types of information that describe a person:

- Name
- Gender
- Birth date
- Address
- Favorite foods

This is certainly not an exhaustive list, but it's good enough for now. The next step is to assign column names and data types. Table 2-7 shows my initial attempt.

Table 2-7. Person table, first pass

Column	Type	Allowable values
`Name`	`Varchar(40)`	
`Gender`	`Char(1)`	`M`, `F`
`Birth_date`	`Date`	
`Address`	`Varchar(100)`	
`Favorite_foods`	`Varchar(200)`	

The `name`, `address`, and `favorite_foods` columns are of type `varchar` and allow for free-form data entry. The `gender` column allows a single character which should equal only `M` or `F`. The `birth_date` column is of type `date`, since a time component is not needed.

Step 2: Refinement

In Chapter 1, you were introduced to the concept of *normalization*, which is the process of ensuring that there are no duplicate (other than foreign keys) or compound columns in your database design. In looking at the columns in the `person` table a second time, the following issues arise:

- The `name` column is actually a compound object consisting of a first name and a last name.
- Since multiple people can have the same name, gender, birth date, and so forth, there are no columns in the `person` table that guarantee uniqueness.
- The `address` column is also a compound object consisting of street, city, state/province, country, and postal code.
- The `favorite_foods` column is a list containing `0`, `1`, or more independent items. It would be best to create a separate table for this data that includes a foreign key to the `person` table so that you know to which person a particular food may be attributed.

After taking these issues into consideration, Table 2-8 gives a normalized version of the `person` table.

Table 2-8. Person table, second pass

Column	Type	Allowable values
Person_id	Smallint (unsigned)	
First_name	Varchar(20)	
Last_name	Varchar(20)	
Gender	Char(1)	M, F
Birth_date	Date	
Street	Varchar(30)	
City	Varchar(20)	
State	Varchar(20)	
Country	Varchar(20)	
Postal_code	Varchar(20)	

Now that the `person` table has a primary key (`person_id`) to guarantee uniqueness, the next step is to build a `favorite_food` table that includes a foreign key to the `person` table. Table 2-9 shows the result.

Table 2-9. Favorite_food table

Column	Type
Person_id	Smallint (unsigned)
Food	Varchar(20)

The `person_id` and `food` columns comprise the primary key of the `favorite_food` table, and the `person_id` column is also a foreign key to the `person` table.

How Much Is Enough?

Moving the `favorite_foods` column out of the `person` table was definitely a good idea, but are we done yet? What happens, for example, if one person lists "pasta" as a favorite food while another person lists "spaghetti"? Are they the same thing? In order to prevent this problem, you might decide that you want people to choose their favorite foods from a list of options, in which case you should create a `food` table with `food_id` and `food_name` columns, and then change the `favorite_food` table to contain a foreign key to the `food` table. While this design would be fully normalized, you might decide that you simply want to store the values that the user has entered, in which case you may leave the table as is.

Step 3: Building SQL Schema Statements

Now that the design is complete for the two tables holding information about people and their favorite foods, the next step is to generate SQL statements to create the tables in the database. Here is the statement to create the `person` table:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender CHAR(1),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Everything in this statement should be fairly self-explanatory except for the last item; when you define your table, you need to tell the database server what column or columns will serve as the primary key for the table. You do this by creating a *constraint* on the table. You can add several types of constraints to a table definition. This constraint is a *primary key constraint*. It is created on the `person_id` column and given the name `pk_person`.

While on the topic of constraints, there is another type of constraint that would be useful for the person table. In Table 2-7, I added a third column to show the allowable values for certain columns (such as 'M' and 'F' for the gender column). Another type of constraint called a *check constraint* constrains the allowable values for a particular column. MySQL allows a check constraint to be attached to a column definition, as in the following:

```
gender CHAR(1) CHECK (gender IN ('M','F')),
```

While check constraints operate as expected on most database servers, the MySQL server allows check constraints to be defined but does not enforce them. However, MySQL does provide another character data type called enum that merges the check constraint into the data type definition. Here's what it would look like for the gender column definition:

```
gender ENUM('M','F'),
```

Here's how the person table definition looks with an enum data type for the gender column:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender ENUM('M','F'),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Later in this chapter, you will see what happens if you try to add data to a column that violates its check constraint (or, in the case of MySQL, its enumeration values).

You are now ready to run the create table statement using the mysql command-line tool. Here's what it looks like:

```
mysql> CREATE TABLE person
    ->  (person_id SMALLINT UNSIGNED,
    ->   fname VARCHAR(20),
    ->   lname VARCHAR(20),
    ->   gender ENUM('M','F'),
    ->   birth_date DATE,
    ->   street VARCHAR(30),
    ->   city VARCHAR(20),
    ->   state VARCHAR(20),
    ->   country VARCHAR(20),
    ->   postal_code VARCHAR(20),
    ->   CONSTRAINT pk_person PRIMARY KEY (person_id)
    ->  );
 Query OK, 0 rows affected (0.27 sec)
```

After processing the `create table` statement, the MySQL server returns the message "Query OK, 0 rows affected," which tells me that the statement had no syntax errors. If you want to make sure that the `person` table does, in fact, exist, you can use the `describe` command (or `desc` for short) to look at the table definition:

```
mysql> DESC person;
+-------------+----------------------+------+-----+---------+-------+
| Field       | Type                 | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| fname       | varchar(20)          | YES  |     | NULL    |       |
| lname       | varchar(20)          | YES  |     | NULL    |       |
| gender      | enum('M','F')        | YES  |     | NULL    |       |
| birth_date  | date                 | YES  |     | NULL    |       |
| street      | varchar(30)          | YES  |     | NULL    |       |
| city        | varchar(20)          | YES  |     | NULL    |       |
| state       | varchar(20)          | YES  |     | NULL    |       |
| country     | varchar(20)          | YES  |     | NULL    |       |
| postal_code | varchar(20)          | YES  |     | NULL    |       |
+-------------+----------------------+------+-----+---------+-------+
10 rows in set (0.06 sec)
```

Columns 1 and 2 of the `describe` output are self-explanatory. Column 3 shows whether a particular column can be omitted when data is inserted into the table. I purposefully left this topic out of the discussion for now (see the sidebar "What Is Null?" on page 29 for a short discourse), but we explore it fully in Chapter 4. The fourth column shows whether a column takes part in any keys (primary or foreign); in this case, the `person_id` column is marked as the primary key. Column 5 shows whether a particular column will be populated with a default value if you omit the column when inserting data into the table. The `person_id` column shows a default value of `0`, although this would work only once, since each row in the `person` table must contain a unique value for this column (since it is the primary key). The sixth column (called "Extra") shows any other pertinent information that might apply to a column.

What Is Null?

In some cases, it is not possible or applicable to provide a value for a particular column in your table. For example, when adding data about a new customer order, the `ship_date` column cannot yet be determined. In this case, the column is said to be *null* (note that I do not say that it *equals* null), which indicates the absence of a value. Null is used for various cases where a value cannot be supplied, such as:

- Not applicable
- Unknown
- Empty set

When designing a table, you may specify which columns are allowed to be null (the default), and which columns are not allowed to be null (designated by adding the keywords `not null` after the type definition).

Now that you've created the `person` table, your next step is to create the `favorite_food` table:

```
mysql> CREATE TABLE favorite_food
    ->  (person_id SMALLINT UNSIGNED,
    ->   food VARCHAR(20),
    ->   CONSTRAINT pk_favorite_food PRIMARY KEY (person_id, food),
    ->   CONSTRAINT fk_fav_food_person_id FOREIGN KEY (person_id)
    ->     REFERENCES person (person_id)
    ->  );
Query OK, 0 rows affected (0.10 sec)
```

This should look very similar to the `create table` statement for the `person` table, with the following exceptions:

- Since a person can have more than one favorite food (which is the reason this table was created in the first place), it takes more than just the `person_id` column to guarantee uniqueness in the table. This table, therefore, has a two-column primary key: `person_id` and `food`.
- The `favorite_food` table contains another type of constraint called a *foreign key constraint*. This constrains the values of the `person_id` column in the `favorite_food` table to include *only* values found in the `person` table. With this constraint in place, I will not be able to add a row to the `favorite_food` table indicating that `person_id 27` likes pizza if there isn't already a row in the `person` table having a `person_id` of `27`.

If you forget to create the foreign key constraint when you first create the table, you can add it later via the `alter table` statement.

`Describe` shows the following after executing the `create table` statement:

```
mysql> DESC favorite_food;
+-------------+---------------------+------+-----+---------+-------+
| Field       | Type                | Null | Key | Default | Extra |
+-------------+---------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| food        | varchar(20)         |      | PRI |         |       |
+-------------+---------------------+------+-----+---------+-------+
```

Now that the tables are in place, the next logical step is to add some data.

Populating and Modifying Tables

With the `person` and `favorite_food` tables in place, you can now begin to explore the four SQL data statements: `insert`, `update`, `delete`, and `select`.

Inserting Data

Since there is not yet any data in the `person` and `favorite_food` tables, the first of the four SQL data statements to be explored will be the `insert` statement. There are three main components to an `insert` statement:

- The name of the table into which to add the data
- The names of the columns in the table to be populated
- The values with which to populate the columns

You are not required to provide data for every column in the table (unless all the columns in the table have been defined as `not null`). In some cases, those columns that are not included in the initial `insert` statement will be given a value later via an `update` statement. In other cases, a column may never receive a value for a particular row of data (such as a customer order that is canceled before being shipped, thus rendering the `ship_date` column inapplicable).

Generating numeric key data

Before inserting data into the `person` table, it would be useful to discuss how values are generated for numeric primary keys. Other than picking a number out of thin air, you have a couple of options:

- Look at the largest value currently in the table and add one.
- Let the database server provide the value for you.

Although the first option may seem valid, it proves problematic in a multiuser environment, since two users might look at the table at the same time and generate the same value for the primary key. Instead, all database servers on the market today provide a safe, robust method for generating numeric keys. In some servers, such as the Oracle Database, a separate schema object is used (called a *sequence*); in the case of MySQL, however, you simply need to turn on the *auto-increment* feature for your primary key column. Normally, you would do this at table creation, but doing it now provides the opportunity to learn another SQL schema statement, `alter table`, which is used to modify the definition of an existing table:

```
ALTER TABLE person MODIFY person_id SMALLINT UNSIGNED AUTO_INCREMENT;
```

This statement essentially redefines the `person_id` column in the `person` table. If you describe the table, you will now see the auto-increment feature listed under the "Extra" column for `person_id`:

```
mysql> DESC person;
+-------------+--------------------------+------+-----+---------+----------------+
| Field       | Type                     | Null | Key | Default | Extra          |
+-------------+--------------------------+------+-----+---------+----------------+
| person_id   | smallint(5) unsigned     |      | PRI | NULL    | auto_increment |
| .           |                          |      |     |         |                |
| .           |                          |      |     |         |                |
| .           |                          |      |     |         |                |
```

When you insert data into the `person` table, simply provide a `null` value for the `person_id` column, and MySQL will populate the column with the next available number (by default, MySQL starts at 1 for auto-increment columns).

The insert statement

Now that all the pieces are in place, it's time to add some data. The following statement creates a row in the `person` table for William Turner:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (null, 'William','Turner', 'M', '1972-05-27');
Query OK, 1 row affected (0.01 sec)
```

The feedback ("Query OK, 1 row affected") tells you that your statement syntax was proper, and that one row was added to the database (since it was an `insert` statement). You can look at the data just added to the table by issuing a `select` statement:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.06 sec)
```

As you can see, the MySQL server generated a value of `1` for the primary key. Since there is only a single row in the `person` table, I neglected to specify which row I am interested in and simply retrieved all the rows in the table. If there were more than one row in the table, however, I could add a `where` clause to specify that I want to retrieve data only for the row having a value of `1` for the `person_id` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE person_id = 1;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

While this query specifies a particular primary key value, you can use any column in the table to search for rows, as shown by the following query, which finds all rows with a value of `'Turner'` for the `lname` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE lname = 'Turner';
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

Before moving on, a couple of things about the earlier `insert` statement are worth mentioning:

- Values were not provided for any of the address columns. This is fine, since `null`s are allowed for those columns.
- The value provided for the `birth_date` column was a string. As long as you match the required format shown in Table 2-5, MySQL will convert the string to a date for you.
- The column names and the values provided must correspond in number and type. If you name seven columns and provide only six values, or if you provide values that cannot be converted to the appropriate data type for the corresponding column, you will receive an error.

William has also provided information about his favorite three foods, so here are three `insert` statements to store his food preferences:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'pizza');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'cookies');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'nachos');
Query OK, 1 row affected (0.01 sec)
```

Here's a query that retrieves William's favorite foods in alphabetical order using an `order by` clause:

```
mysql> SELECT food
    -> FROM favorite_food
    -> WHERE person_id = 1
    -> ORDER BY food;
+---------+
| food    |
+---------+
| cookies |
| nachos  |
| pizza   |
```

```
+---------+
3 rows in set (0.02 sec)
```

The `order by` clause tells the server how to sort the data returned by the query. Without the `order by` clause, there is no guarantee that the data in the table will be retrieved in any particular order.

So that William doesn't get lonely, you can execute another `insert` statement to add Susan Smith to the `person` table:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date,
    ->   street, city, state, country, postal_code)
    -> VALUES (null, 'Susan','Smith', 'F', '1975-11-02',
    ->   '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
 Query OK, 1 row affected (0.01 sec)
```

Since Susan was kind enough to provide her address, we included five more columns than when William's data was inserted. If you query the table again, you will see that Susan's row has been assigned the value `2` for its primary key value:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
|         2 | Susan   | Smith  | 1975-11-02 |
+-----------+---------+--------+------------+
2 rows in set (0.00 sec)
```

Can I Get That in XML?

If you will be working with XML data, you will be happy to know that most database servers provide a simple way to generate XML output from a query. With MySQL, for example, you can use the `--xml` option when invoking the `mysql` tool, and all your output will automatically be formatted using XML. Here's what the favorite-food data looks like as an XML document:

```
C:\database> mysql -u lrngsql -p --xml bank
Enter password: xxxxxx
Welcome to the MySQL Monitor...

Mysql> SELECT * FROM favorite_food;
<?xml version="1.0"?>

<resultset statement="select * from favorite_food"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="person_id">1</field>
        <field name="food">cookies</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">nachos</field>
  </row>
```

```
  <row>
        <field name="person_id">1</field>
        <field name="food">pizza</field>
  </row>
</resultset>
3 rows in set (0.00 sec)
```

With SQL Server, you don't need to configure your command-line tool; you just need to add the `for xml` clause to the end of your query, as in:

```
SELECT * FROM favorite_food
FOR XML AUTO, ELEMENTS
```

Updating Data

When the data for William Turner was initially added to the table, data for the various address columns was omitted from the `insert` statement. The next statement shows how these columns can be populated via an `update` statement:

```
mysql> UPDATE person
    -> SET street = '1225 Tremont St.',
    ->   city = 'Boston',
    ->   state = 'MA',
    ->   country = 'USA',
    ->   postal_code = '02138'
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The server responded with a two-line message: the "Rows matched: 1" item tells you that the condition in the `where` clause matched a single row in the table, and the "Changed: 1" item tells you that a single row in the table has been modified. Since the `where` clause specifies the primary key of William's row, this is exactly what you would expect to have happen.

Depending on the conditions in your `where` clause, it is also possible to modify more than one row using a single statement. Consider, for example, what would happen if your `where` clause looked as follows:

```
WHERE person_id < 10
```

Since both William and Susan have a `person_id` value less than 10, both of their rows would be modified. If you leave off the `where` clause altogether, your `update` statement will modify every row in the table.

Deleting Data

It seems that William and Susan aren't getting along very well together, so one of them has got to go. Since William was there first, Susan will get the boot courtesy of the `delete` statement:

```
mysql> DELETE FROM person
    -> WHERE person_id = 2;
Query OK, 1 row affected (0.01 sec)
```

Again, the primary key is being used to isolate the row of interest, so a single row is deleted from the table. Similar to the `update` statement, more than one row can be deleted depending on the conditions in your `where` clause, and all rows will be deleted if the `where` clause is omitted.

When Good Statements Go Bad

So far, all of the SQL data statements shown in this chapter have been well formed and have played by the rules. Based on the table definitions for the `person` and `favorite_food` tables, however, there are lots of ways that you can run afoul when inserting or modifying data. This section shows you some of the common mistakes that you might come across and how the MySQL server will respond.

Nonunique Primary Key

Because the table definitions include the creation of primary key constraints, MySQL will make sure that duplicate key values are not inserted into the tables. The next statement attempts to bypass the auto-increment feature of the `person_id` column and create another row in the `person` table with a `person_id` of `1`:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (1, 'Charles','Fulton', 'M', '1968-01-15');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

There is nothing stopping you (with the current schema objects, at least) from creating two rows with identical names, addresses, birth dates, and so on, as long as they have different values for the `person_id` column.

Nonexistent Foreign Key

The table definition for the `favorite_food` table includes the creation of a foreign key constraint on the `person_id` column. This constraint ensures that all values of `person_id` entered into the `favorite_food` table exist in the `person` table. Here's what would happen if you tried to create a row that violates this constraint:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (999, 'lasagna');
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails ('bank'.'favorite_food', CONSTRAINT 'fk_fav_food_person_id' FOREIGN KEY
('person_id') REFERENCES 'person' ('person_id'))
```

In this case, the `favorite_food` table is considered the *child* and the `person` table is considered the *parent*, since the `favorite_food` table is dependent on the `person` table

for some of its data. If you plan to enter data into both tables, you will need to create a row in `parent` before you can enter data into `favorite_food`.

Foreign key constraints are enforced only if your tables are created using the InnoDB storage engine. We discuss MySQL's storage engines in Chapter 12.

Column Value Violations

The `gender` column in the `person` table is restricted to the values `'M'` for male and `'F'` for female. If you mistakenly attempt to set the value of the column to any other value, you will receive the following response:

```
mysql> UPDATE person
    -> SET gender = 'Z'
    -> WHERE person_id = 1;
ERROR 1265 (01000): Data truncated for column 'gender' at row 1
```

The error message is a bit confusing, but it gives you the general idea that the server is unhappy about the value provided for the `gender` column.

Invalid Date Conversions

If you construct a string with which to populate a `date` column, and that string does not match the expected format, you will receive another error. Here's an example that uses a date format that does not match the default date format of "YYYY-MM-DD":

```
mysql> UPDATE person
    -> SET birth_date = 'DEC-21-1980'
    -> WHERE person_id = 1;
ERROR 1292 (22007): Incorrect date value: 'DEC-21-1980' for column 'birth_date'
at row 1
```

In general, it is always a good idea to explicitly specify the format string rather than relying on the default format. Here's another version of the statement that uses the `str_to_date` function to specify which format string to use:

```
mysql> UPDATE person
    -> SET birth_date = str_to_date('DEC-21-1980' , '%b-%d-%Y')
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Not only is the database server happy, but William is happy as well (we just made him eight years younger, without the need for expensive cosmetic surgery!).

Earlier in the chapter, when I discussed the various temporal data types, I showed date-formatting strings such as "YYYY-MM-DD". While many database servers use this style of formatting, MySQL uses `%Y` to indicate a four-character year. Here are a few more formatters that you might need when converting strings to `datetime`s in MySQL:

```
%a The short weekday name, such as Sun, Mon, ...
%b The short month name, such as Jan, Feb, ...
%c The numeric month (0..12)
%d The numeric day of the month (00..31)
%f The number of microseconds (000000..999999)
%H The hour of the day, in 24-hour format (00..23)
%h The hour of the day, in 12-hour format (01..12)
%i The minutes within the hour (00..59)
%j The day of year (001..366)
%M The full month name (January..December)
%m The numeric month
%p AM or PM
%s The number of seconds (00..59)
%W The full weekday name (Sunday..Saturday)
%w The numeric day of the week (0=Sunday..6=Saturday)
%Y The four-digit year
```

The Bank Schema

For the remainder of the book, you use a group of tables that model a community bank. Some of the tables include `Employee`, `Branch`, `Account`, `Customer`, `Product`, and `Transaction`. The entire schema and example data should have been created when you followed the final steps at the beginning of the chapter for loading the MySQL server and generating the sample data. To see a diagram of the tables and their columns and relationships, see Appendix A.

Table 2-10 shows all the tables used in the bank schema along with short definitions.

Table 2-10. Bank schema definitions

Table name	Definition
`Account`	A particular product opened for a particular customer
`Branch`	A location at which banking transactions are conducted
`Business`	A corporate customer (subtype of the `Customer` table)
`Customer`	A person or corporation known to the bank
`Department`	A group of bank employees implementing a particular banking function
`Employee`	A person working for the bank
`Individual`	A noncorporate customer (subtype of the `Customer` table)
`Officer`	A person allowed to transact business for a corporate customer
`Product`	A banking service offered to customers
`Product_type`	A group of products having a similar function
`Transaction`	A change made to an account balance

Feel free to experiment with the tables as much as you want, including adding your own tables to expand the bank's business functions. You can always drop the database and re-create it from the downloaded file if you want to make sure your sample data is intact.

If you want to see the tables available in your database, you can use the `show tables` command, as in:

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_bank |
+----------------+
| account        |
| branch         |
| business       |
| customer       |
| department     |
| employee       |
| favorite_food  |
| individual     |
| officer        |
| person         |
| product        |
| product_type   |
| transaction    |
+----------------+
13 rows in set (0.10 sec)
```

Along with the 11 tables in the bank schema, the table listing also includes the two tables created in this chapter: `person` and `favorite_food`. These tables will not be used in later chapters, so feel free to drop them by issuing the following commands:

```
mysql> DROP TABLE favorite_food;
Query OK, 0 rows affected (0.56 sec)
mysql> DROP TABLE person;
Query OK, 0 rows affected (0.05 sec)
```

If you want to look at the columns in a table, you can use the `describe` command. Here's an example of the `describe` output for the `customer` table:

```
mysql> DESC customer;
+-------------+------------------+------+-----+---------+----------------+
| Field       | Type             | Null | Key | Default | Extra          |
+-------------+------------------+------+-----+---------+----------------+
| cust_id     | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| fed_id      | varchar(12)      | NO   |     | NULL    |                |
| cust_type_cd| enum('I','B')    | NO   |     | NULL    |                |
| address     | varchar(30)      | YES  |     | NULL    |                |
| city        | varchar(20)      | YES  |     | NULL    |                |
| state       | varchar(20)      | YES  |     | NULL    |                |
| postal_code | varchar(10)      | YES  |     | NULL    |                |
+-------------+------------------+------+-----+---------+----------------+
7 rows in set (0.03 sec)
```

The more comfortable you are with the example database, the better you will understand the examples and, consequently, the concepts in the following chapters.