5
MySQL

5.1 INTRODUCTION

Many of the applications that a Web developer wants to use can be made easier by the use of a standardized database to store, organize, and access information. MySQL is an Open Source (GPL) Standard Query Language (SQL) database that is fast, reliable, easy to use, and suitable for applications of any size. SQL is the ANSI-standard database query language used by most databases (though all have their nonstandard extensions).

MySQL can easily be integrated into Perl programs by using the Perl DBI (DataBase Independent interface) module. DBI is an Application Program Interface (API) that allows Perl to connect to and query a number of SQL databases (among them MySQL, mSQL, PostgreSQL, Oracle, Sybase, and Informix).

If you installed Linux as suggested in Chapter 2, MySQL and DBI are already installed.

5.2 TUTORIAL

Following the Swiss Army knife theory (20 percent of the functions give you 80 percent of the utility), a few SQL commands go a long way to facilitate learning MySQL/Perl/DBI.

To illustrate these, we create a simple database containing information about some (fictional) people. Eventually, we'll show how to enter this information from a form on the Web (see Chapter 7), but for now we interface with SQL directly.

First, try to make a connection to our MySQL server as the `root` MySQL user:

```
$ mysql -u root
```

N The MySQL `root` user is different from the Linux `root` user. The MySQL `root` user is used to administer the MySQL server only.

If you see the following output:

```
ERROR 2002: Can't connect to local MySQL server through socket
´/var/lib/mysql/mysql.sock´(2)
```

it likely means the MySQL server is not running. If your system is set up securely, it shouldn't be running, because you had no reason, before now, for it to be running. Use `chkconfig` as `root` to make sure it starts the next time the machine boots, and then start it by hand as follows:

```
# chkconfig mysqld on
# /etc/init.d/mysqld start
```

Now you should be able to connect (*not* logged in as the Linux `root` user):

```
$ mysql -u root
```

If not, see the MySQL log file at `/var/log/mysqld.log`. If so, you'll see a welcome message and the MySQL prompt:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.23.36

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

mysql>
```

As suggested, enter `help;` at the prompt. A list of MySQL commands (not to be confused with SQL commands) will be displayed. These allow you to work with the MySQL server. For grins, enter `status;` to see the status of the server.

To illustrate these commands, we will create a database called `people` that contains information about people and their ages.

5.2.1 The SHOW DATABASES and CREATE DATABASE Commands

First, we need to create the new database. Check the current databases to make sure a database of that name doesn't already exist; then create the new one, and verify the existence of the new database:

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.00 sec)

mysql> CREATE DATABASE people;
Query OK, 1 row affected (0.00 sec)

mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| people   |
| test     |
+----------+
3 rows in set (0.00 sec)
```

SQL commands and subcommands (in the previous example, `CREATE` is a command; `DATABASE` is its subcommand) are case-insensitive. The name of the database (and table and field) are case sensitive. It's a matter of style whether one uses uppercase or lowercase, but traditionally the SQL commands are distinguished by uppercase.

One way to think of a *database* is as a container for related *tables*. A table is a collection of *rows,* each row holding data for one *record,* each record containing chunks of information called *fields*.

5.2.2 The USE Command

Before anything can be done with the newly created database, MySQL has to connect to it. That's done with the `USE` command:

```
mysql> USE people;
```

5.2.3 The CREATE TABLE and SHOW TABLES Commands

Each table within the database must be defined and created. This is done with the `CREATE TABLE` command.

Create a table named `age_information` to contain an individual's first name, last name, and age. MySQL needs to know what kind of data can be stored in these fields. In this case, the first name and the last name are character strings of up to 20 characters each, and the age is an integer:

```
mysql> CREATE TABLE age_information (
    -> lastname   CHAR(20),
    -> firstname  CHAR(20),
    -> age        INT
    -> );
Query OK, 0 rows affected (0.00 sec)
```

It appears that the table was created properly (it says OK after all), but this can be checked by executing the `SHOW TABLES` command. If an error is made, the table can be removed with `DROP TABLE`.

When a database in MySQL is created, a directory is created with the same name as the database (`people`, in this example):

```
# ls -l /var/lib/mysql
total 3
drwx------    2 mysql    mysql        1024 Dec 12 15:28 mysql
srwxrwxrwx    1 mysql    mysql           0 Dec 13 07:19 mysql.sock
drwx------    2 mysql    mysql        1024 Dec 13 07:24 people
drwx------    2 mysql    mysql        1024 Dec 12 15:28 test
```

Within that directory, each table is implemented with three files:

```
# ls -l /var/lib/mysql/people
total 10
-rw-rw----    1 mysql    mysql     8618 Dec 13 07:24 age_information.frm
-rw-rw----    1 mysql    mysql        0 Dec 13 07:24 age_information.MYD
-rw-rw----    1 mysql    mysql     1024 Dec 13 07:24 age_information.MYI

mysql> SHOW TABLES;
+------------------+
| Tables_in_people |
+------------------+
| age_information  |
+------------------+
1 row in set (0.00 sec)
```

This example shows two MySQL datatypes: character strings and integers. Other MySQL data types include several types of integers

(for a complete discussion of MySQL's data types, see `www.mysql.com/documentation/mysql/bychapter/manual_Reference.html#Column_types`):

`TINYINT`	−128 to 127 (signed) or 0 to 255 (unsigned)
`SMALLINT`	−32768 to 32767 (signed) or 0 to 65535 (unsigned)
`MEDIUMINT`	−8388608 to 8388607 (signed) or 0 to 16777215 (unsigned)
`INTEGER` (same as `INT`)	−2147483648 to 2147483647 (signed) or 0 to 4294967295 (unsigned)
`BIGINT`	−9223372036854775808 to 9223372036854775807 (signed) or 0 to 18446744073709551615 (unsigned)

and floating points:

`FLOAT`

`DOUBLE`

`REAL` (same as `DOUBLE`)

`DECIMAL`

`NUMERIC` (same as `DECIMAL`)

There are several data types to represent a date:

`DATE`	YYYY-MM-DD
`DATETIME`	YYYY-MM-DD HH:MM:SS
`TIMESTAMP`	YYYYMMDDHHMMSS or YYMMDDHHMMSS or YYYYMMDD or YYMMDD
`TIME`	HH:MM:SS
`YEAR`	YYYY or YY

The table `age_information` used the `CHAR` character data type. The following are the other character data types. Several have `BLOB` in their name—a `BLOB` is a Binary Large OBject that can hold a variable amount of data. The types with `TEXT` in their name are just like their corresponding `BLOB`s

except when matching is involved: The BLOBs are case-sensitive, and the TEXTs are case-insensitive.

VARCHAR	variable-length string up to 255 characters
TINYBLOB TINYTEXT	maximum length 255 characters
BLOB TEXT	maximum length 65535 characters
MEDIUMBLOB MEDIUMTEXT	maximum length 16777215 characters
LONGBLOB LONGTEXT	maximum length 4294967295 characters

5.2.4 The DESCRIBE Command

The DESCRIBE command gives information about the fields in a table. The fields created earlier—lastname, firstname, and age—appear to have been created correctly.

```
mysql> DESCRIBE age_information;
+-----------+----------+------+-----+---------+-------+
| Field     | Type     | Null | Key | Default | Extra |
+-----------+----------+------+-----+---------+-------+
| lastname  | char(20) | YES  |     | NULL    |       |
| firstname | char(20) | YES  |     | NULL    |       |
| age       | int(11)  | YES  |     | NULL    |       |
+-----------+----------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

The command SHOW COLUMNS FROM age_information; gives the same information as DESCRIBE age_information; but DESCRIBE involves less typing. (If you're really trying to save keystrokes, you could abbreviate DESCRIBE as DESC.)

5.2.5 The INSERT Command

For the table to be useful, we need to add information to it. We do so with the INSERT command:

```
mysql> INSERT INTO age_information
    ->       (lastname, firstname, age)
    ->       VALUES (´Wall´, ´Larry´, 46);
Query OK, 1 row affected (0.00 sec)
```

The syntax of the command is `INSERT INTO`, followed by the table in which to insert, a list within parentheses of the fields into which information is to be inserted, and the qualifier `VALUES` followed by the list of values in parentheses in the same order as the respective fields.[1]

5.2.6 The SELECT Command

`SELECT` selects records from the database. When this command is executed from the command line, MySQL prints all the records that match the query. The simplest use of `SELECT` is shown in this example:

```
mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   46 |
+----------+-----------+------+
1 row in set (0.00 sec)
```

The `*` means "show values for all fields in the table"; `FROM` specifies the table from which to extract the information.

The previous output shows that the record for Larry Wall was added successfully. To experiment with the `SELECT` command, we need to add a few more records, just to make things interesting:

```
mysql> INSERT INTO age_information
    ->       (lastname, firstname, age)
    ->       VALUES (´Torvalds´, ´Linus´, 31);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO age_information
    ->       (lastname, firstname, age)
    ->       VALUES (´Raymond´, ´Eric´, 40);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   46 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)
```

[1]We did extensive research to determine that none of the names used in this chapter belong to real people.

There are many ways to use the SELECT command—it's very flexible. First, sort the table based on lastname:

```
mysql> SELECT * FROM age_information
    ->        ORDER BY lastname;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Raymond  | Eric      |   40 |
| Torvalds | Linus     |   31 |
| Wall     | Larry     |   46 |
+----------+-----------+------+
3 rows in set (0.00 sec)
```

Now show only the lastname field, sorted by lastname:

```
mysql> SELECT lastname FROM age_information
    ->        ORDER BY lastname;
+----------+
| lastname |
+----------+
| Raymond  |
| Torvalds |
| Wall     |
+----------+
3 rows in set (0.00 sec)
```

Show the ages in descending order:

```
mysql> SELECT age FROM age_information ORDER BY age DESC;
+------+
| age  |
+------+
|   46 |
|   40 |
|   31 |
+------+
3 rows in set (0.00 sec)
```

Show all the last names for those who are older than 35:

```
mysql> SELECT lastname FROM age_information WHERE age > 35;
+----------+
| lastname |
+----------+
| Wall     |
| Raymond  |
+----------+
2 rows in set (0.00 sec)
```

Do the same, but sort by `lastname`:

```
mysql> SELECT lastname FROM age_information
    ->     WHERE age > 35 ORDER BY lastname;
+----------+
| lastname |
+----------+
| Raymond  |
| Wall     |
+----------+
2 rows in set (0.00 sec)
```

5.2.7 The UPDATE Command

Since the database is about people, information in it can change (people are unpredictable like that). For instance, although a person's birthday is static, their age changes. To change the value in an existing record, we can `UPDATE` the table. Let's say the fictional Larry Wall has turned 47:

```
mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   46 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)

mysql> UPDATE age_information SET age = 47
    ->       WHERE lastname = 'Wall';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   47 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)
```

Be sure to use that `WHERE` clause; otherwise, if we had only entered `UPDATE age_information SET age = 47`, all the records in the database would have been given the age of 47!

Although this might be good news for some people in these records (how often have the old-timers said "Oh, to be 47 years old again"—OK, probably not), it might be shocking news to others.

This method works, but it requires the database to know that Larry is 46, turning 47. Instead of keeping track of this, for Larry's next birthday we simply increment his age:

```
mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   47 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)

mysql> UPDATE age_information SET age = age + 1
    ->        WHERE lastname = 'Wall';

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   48 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)
```

5.2.8 The DELETE Command

Sometimes we need to delete a record from the table (don't assume the worst—perhaps the person just asked to be removed from a mailing list, which was opt-in in the first place, of course). This is done with the `DELETE` command:

```
mysql> DELETE FROM age_information WHERE lastname = 'Raymond';
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   48 |
| Torvalds | Linus     |   31 |
+----------+-----------+------+
2 rows in set (0.00 sec)
```

Eric is in good company here, so put him back:

```
mysql> INSERT INTO age_information
    ->       (lastname, firstname, age)
    ->       VALUES (´Raymond´, ´Eric´, 40);

Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   48 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)
```

5.2.9 Some Administrative Details

All these examples have been executed as the `root` MySQL user, which, as you might imagine, is not optimal from a security standpoint. A better practice is to create a MySQL user who can create and update tables as needed.

First, as a security measure, change the MySQL `root` password when logging in to the server:

```
# mysqladmin password IAmGod
```

Now when `mysql` executes, a password must be provided using the `-p` switch. Here is what would happen if we forgot the `-p`:

```
$ mysql -u root
ERROR 1045: Access denied for user: ´root@localhost´ (Using password: NO)
```

Try again using `-p`. When prompted for the password, enter the one given previously:

Recall that the MySQL user is *not* the same as a Linux user. The `mysqladmin` command changes the password for the MySQL user only, not the Linux user. For security reasons, we suggest that the MySQL password never be the same as the password used to log in to the Linux machine. Also, the password `IAmGod`, which is clever, is a bad password for many reasons, including the fact that it is used as an example in this book. For a discussion on what makes a password good or bad, we suggest you read *Hacking Linux Exposed* [Hatch+ 02].

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 3.23.36

Type ´help;´ or ´\h´ for help. Type ´\c´ to clear the buffer
mysql>
```

Doing all the SQL queries in the `people` database as the MySQL `root` user is a Bad Idea (see HLE if you want proof of this). So let's create a new user. This involves modifying the database named `mysql`, which contains all the administrative information for the MySQL server, so first we use the `mysql` database and then grant privileges for a new user:

```
mysql> USE mysql;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> GRANT SELECT,INSERT,UPDATE,DELETE
    ->     ON people.*
    ->     TO apache@localhost
    ->     IDENTIFIED BY ´LampIsCool´;
Query OK, 0 rows affected (0.00 sec)
```

The user `apache` (the same user that runs the webserver) is being granted the ability to do most everything within the database, including being able to delete entries in tables within the `people` database. However, `apache` cannot delete the `people` database, only entries within the tables in the database. The user `apache` can access the `people` database from `localhost` only (instead of being able to log in over the network from another machine).

The `IDENTIFIED BY` clause in the SQL command sets the `apache` user's password to `LampIsCool`. Setting the password is necessary only the first

time permissions are granted for this user—later, when the `apache` user is given permissions in other databases, the password doesn't need to be reset.

To verify that these changes were made, log in as `apache`:

```
$ mysql -u apache -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 27 to server version: 3.23.36

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

mysql> USE people
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+------------------+
| Tables_in_people |
+------------------+
| age_information  |
+------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM age_information;
+----------+-----------+------+
| lastname | firstname | age  |
+----------+-----------+------+
| Wall     | Larry     |   48 |
| Torvalds | Linus     |   31 |
| Raymond  | Eric      |   40 |
+----------+-----------+------+
3 rows in set (0.00 sec)
```

5.2.10 Summary

As discussed, these commands are enough to do basic things with MySQL:

```
SHOW DATABASES
CREATE DATABASE
USE
CREATE TABLE
SHOW TABLES
DESCRIBE
INSERT
SELECT
UPDATE
DELETE
GRANT
```

5.3 DATABASE INDEPENDENT INTERFACE

Running MySQL commands from the shell is well and good the first 12 times it has to be done. After that, the typical lazy programmer starts thinking of ways to automate the process. Here, the answer is Perl and the DataBase Independent interface (DBI). DBI enables one to write programs to automate database maintenance and to write other scripts to interface with MySQL.

DBI is a Perl module that provides methods to manipulate SQL databases. With DBI, one can connect to a database within a Perl script and issue all kinds of queries, including `SELECT`, `INSERT`, and `DELETE`. For now, we create Perl scripts that can be run from the shell. Later, we'll use CGI, mod_perl, Embperl, Mason, and PHP to hook database independent interfaces into web programs.

First, a quick example. We put all these DBI examples in a directory that is under `/var/www/` so that the examples are downloadable from `www.opensourcewebbook.com/`. In the real world, we do not suggest you create a directory under `/var/www/` to create arbitrary Perl programs, but for our purposes, it just makes life easier when downloading all the examples. Create the directory and go there:

```
$ mkdir /var/www/bin
$ cd /var/www/bin
```

The first example demonstrates how to connect to a database. This code is stored in the file `/var/www/bin/connect.pl` and online at `http://localhost/mysql/connect.pl` or `www.opensourcewebbook.com/mysql/connect.pl`. The content of `connect.pl` is:

```
#!/usr/bin/perl -w
# connect.pl

# use the DBI module
use DBI;

# use strict, it is a Good Idea
use strict;

# connect to the database, assigning the result to $dbh
my $dbh = DBI->connect(´DBI:mysql:people´, ´apache´, ´LampIsCool´);
```

```
# die if we failed to connect
die "Can't connect: " . DBI->errstr() unless $dbh;

# all is well!
print "Success: connected!\n";

# disconnect from the MySQL server
$dbh->disconnect();
```

First, the `use DBI` method tells Perl to use the DBI module. This allows us to use all the methods in this class.

Calling the `connect()` method causes the Perl script to connect to the MySQL database using the Perl DBI class. The first argument to this method is the database to which you want to connect. In this example, the string `DBI:mysql:people` indicates that it should connect with the DBI module to the database `people`, which is housed on the local MySQL server. The second and third arguments to the `connect()` method are the username and password used to connect. Here user `apache` and the supersecret password are passed. If successful, `connect()` returns a *database handle* that is assigned to `$dbh`.

> If one day we decide that we want to migrate to another database, such as Oracle, we merely need to change `mysql` to `oracle`, and the rest of the script stays exactly the same, assuming the script is not executing a query that is specific to that database server—certainly the case with the scripts in this book. Design for portability!

If `connect()` returns false, the script `die`s, printing the error string returned by the `errstr()` method. If the script doesn't `die`, it prints a message stating that all is well. This gives us a warm, fuzzy feeling (for maximum fuzzy feeling, perhaps we should have printed "hello, world").

The last thing done is to execute the `disconnect()` method, allowing the Perl script and database to properly shut down the connection. This is only polite, and if you don't call `disconnect()`, the script may generate an error message, and the MySQL server will not like you.

Executing this program from the shell produces:

```
$ ./connect.pl
Success: connected!
```

We've connected. But by itself, connecting isn't exceptionally useful, so let's see what records are in the `age_information` table. Create (or download) the script `/var/www/bin/show_ages.pl`. Online, it is at `http://localhost/mysql/show_ages.pl` or `www.opensourcewebbook.com/mysql/show_ages.pl`. Its contents are as follows:

```
#!/usr/bin/perl -w
# show_ages.pl

use DBI;

use strict;

# connect to the server, and if connect returns false,
# die() with the DBI error string
my $dbh = DBI->connect(´DBI:mysql:people´, ´apache´, ´LampIsCool´)
        or die "Can't connect: " . DBI->errstr();

# prepare the SQL, die() if the preparation fails
my $sth = $dbh->prepare(´SELECT * FROM age_information´)
        or die "Can't prepare SQL: " . $dbh->errstr();

# execute the SQL, die() if it fails
$sth->execute()
        or die "Can't execute SQL: " . $sth->errstr();

# loop through each record of our table,
# $sth->fetchrow() returns the next row,
# and we store the values in $ln, $fn and $age
my($ln, $fn, $age);
while (($ln, $fn, $age) = $sth->fetchrow()) {
    print "$fn $ln, $age\n";
}

# finish the statement handle, disconnect from the server
$sth->finish();
$dbh->disconnect();
```

Failure to connect is handled differently by this program. It executes `connect()` and uses the `or` to mimic an `unless`. If the `connect()` fails, the script `die`s.

The script then prepares the SQL query `"SELECT * FROM age_information"`. The query is just like that we might have typed into the MySQL program in

the earlier examples (except the command terminator ; is not required in the `prepare()` method). The `prepare()` method returns a *statement handle* object that can then be used to execute the SQL query by calling the `execute()` method. Note that with each of these calls, failure is handled with the `or die()` code.

The results of the `SELECT` query are handled with a `while` loop. The `fetchrow()` method returns a list of data for the next row of data that is returned by the query, which is then assigned to `$ln` (last name), `$fn` (first name), and `$age`. The information is then printed.

At the end, the `finish()` method is executed to properly clean up and because it is the right thing to do. Running this from the shell produces:

```
$ ./show_ages.pl
Larry Wall, 48
Linus Torvalds, 31
Eric Raymond, 40
```

How might we enter a new record into the table? This code is in the file `/var/www/bin/insert.pl`. The entire contents of this program can be found online at `http://localhost/mysql/insert.pl` or `www.opensourcewebbook.com/mysql/insert.pl`. Here is the good part:

```
# print a nice dashed line
print ´-´ x 40, "\n\n";

# now, prompt for and read in the data for the new record
print ´Enter last name: ´;
chomp($ln = <STDIN>);
print ´Enter first name: ´;
chomp($fn = <STDIN>);
print ´Enter age: ´;
chomp($age = <STDIN>);

# prepare SQL for insert
$sth = $dbh->prepare(´INSERT INTO age_information
                (
                    lastname,
                    firstname,
                    age
                )
            VALUES
                (
                    ?,
                    ?,
                    ?
                )´)
        or die "Can't prepare SQL: " . $dbh->errstr();
```

```
# insert the record - note the arguments to execute()
$sth->execute($ln, $fn, $age)
        or die "Can't execute SQL: " . $sth->errstr();

# print another dashed line
print "\n", '-' x 40, "\n\n";
```

Before new data is inserted into the table, the script connects to the server and shows the current contents, just as in `show_ages.pl`.

Then the script asks the user to enter the last name, first name, and age of the person for the new record and `chomp()`s the newlines.

> Be sure to use those question marks as placeholders. This prevents the need to escape quotes and other nasty characters, thus making the code more secure. Also, in this case, the last name is defined in the tables as 20 characters of text. If the user enters more than 20 characters, only the first 20 are used—hence, no overflow problem (although it wouldn't hurt to double-check the length of the input strings).

The next step is to prepare SQL for the `INSERT` query. Again, it looks much like what one would have typed in directly to SQL, with whitespace characters for readability, except that it has those three question marks. Those question marks are placeholders for the contents of the variables in the `execute()` method. The variables `$ln`, `$fn`, and `$age` are inserted into the query where the question marks are, in that order.

To check that the insert worked, the script displays the contents of the table after the `INSERT` is executed. Then the script cleans up after itself by finishing the statement handle and disconnecting from the MySQL server.

Executing that code produces:

```
$ ./insert.pl
Larry Wall, 48
Linus Torvalds, 31
Eric Raymond, 40
----------------------------------------
Enter last name: Ballard
Enter first name: Ron
Enter age: 31

----------------------------------------
```

```
Larry Wall, 48
Linus Torvalds, 31
Eric Raymond, 40
Ron Ballard, 31
```

5.4 TABLE JOINS

In the world of relational databases, data often has complex relationships and is spread across multiple tables. Sometimes it is necessary to grab information from one table based on information in another. This requires that the two tables be `JOIN`ed.

For an example, we create a new table in the `people` database called `addresses` that contains information about people's addresses (surprise!). First, it must be created as follows:

```
mysql> CREATE TABLE addresses (
    ->     lastname CHAR(20),
    ->     firstname CHAR(20),
    ->     address CHAR(40),
    ->     city CHAR(20),
    ->     state CHAR(2),
    ->     zip CHAR(10)
    -> );
```

The table needs some data:

```
mysql> INSERT INTO addresses
    ->    (lastname, firstname, address, city, state, zip)
    ->    VALUES ("Wall", "Larry", "Number 1 Perl Way",
    ->            "Cupertino", "CA", "95015-0189"
    -> );
mysql> INSERT INTO addresses
    ->    (lastname, firstname, address, city, state, zip)
    ->    VALUES ("Torvalds", "Linus", "123 Main St.",
    ->            "San Francisco", "CA", "94109-1234"
    -> );
mysql> INSERT INTO addresses
    ->    (lastname, firstname, address, city, state, zip)
    ->    VALUES ("Raymond", "Eric", "987 Oak St.",
    ->            "Chicago", "IL", "60601-4510"
    -> );
mysql> INSERT INTO addresses
    ->    (lastname, firstname, address, city, state, zip)
    ->    VALUES ("Kedzierski", "John", "3492 W. 75th St.",
    ->            "New York", "NY", "10010-1010"
    -> );
```

```
mysql> INSERT INTO addresses
    ->    (lastname, firstname, address, city, state, zip)
    ->    VALUES ("Ballard", "Ron", "4924 Chicago Ave.",
    ->            "Evanston", "IL", "60202-0440"
    -> );
```

To verify the tables were populated, do this:

```
mysql> SELECT * FROM age_information;
+------------+-----------+------+
| lastname   | firstname | age  |
+------------+-----------+------+
| Wall       | Larry     |   46 |
| Torvalds   | Linus     |   31 |
| Raymond    | Eric      |   40 |
| Kedzierski | John      |   23 |
| Ballard    | Ron       |   31 |
+------------+-----------+------+
5 rows in set (0.00 sec)

mysql> SELECT * FROM addresses;
+----------+---------+-----------------+--------------+-----+----------+
|lastname  |firstname|address          |city          |state|zip       |
+----------+---------+-----------------+--------------+-----+----------+
|Wall      |Larry    |# 1 Perl Way     |Cupertino     |CA   |95015-0189|
|Torvalds  |Linus    |123 Main St.     |San Francisco |CA   |94109-1234|
|Raymond   |Eric     |987 Oak St.      |Chicago       |IL   |60601-4510|
|Kedzierski|John     |3492 W. 75th St. |New York      |NY   |10010-1010|
|Ballard   |Ron      |4924 Chicago Ave.|Evanston      |IL   |60202-0440|
+----------+---------+-----------------+--------------+-----+----------+
5 rows in set (0.00 sec)
```

Now, on to the `JOIN`s. Let's say we want to find out what city our under-40-year-old people live in. This requires looking up information in two tables: To find out who is under 40, we look in `age_information`, and to find out the city, we look in `addresses`. Therefore, we need to tell the `SELECT` command about both tables.

Because both tables are being used, we need to be specific about which table a particular field belongs to. In other words, instead of saying `SELECT city`, we need to say what table that field is in, so we say `SELECT addresses.city`. The `addresses.city` tells MySQL that the table is `addresses` and the field is `city`.

Moreover, we need to hook the two tables together somehow—we do so with the following command by making sure the `lastname` from the

addresses row matches the lastname from the age_information row. Ditto for the firstname. So, our command is:

```
mysql> SELECT addresses.city
    ->    FROM addresses, age_information
    ->    WHERE age_information.age < 40 AND
    ->          addresses.lastname = age_information.lastname
    ->      AND addresses.firstname = age_information.firstname;
+---------------+
| city          |
+---------------+
| San Francisco |
| New York      |
| Evanston      |
+---------------+
3 rows in set (0.02 sec)
```

In English, we are saying, "give me the city for all the people with ages less than 40, where the last names and first names match in each row."

Let's grab the last names and zip codes for all those 40 and over, and order the data based on the last name:

```
mysql> SELECT addresses.lastname, addresses.zip
    ->     FROM addresses, age_information
    ->     WHERE age_information.age >= 40 AND
    ->           addresses.lastname = age_information.lastname AND
    ->           addresses.firstname = age_information.firstname
    ->     ORDER BY addresses.lastname;
+----------+------------+
| lastname | zip        |
+----------+------------+
| Raymond  | 60601-4510 |
| Wall     | 95015-0189 |
+----------+------------+
2 rows in set (0.02 sec)
```

As you can see, there are lots of different ways to query more than one table to get the exact information desired.

5.5 LOADING AND DUMPING A DATABASE

We can load a database or otherwise execute SQL commands from a file. We simply put the commands or database into a file—let's call it mystuff.sql—and load it in with this command:

```
$ mysql people < mystuff.sql
```

We can also dump out a database into a file with this command:

```
$ mysqldump people > entiredb.sql
```

For fun, try the `mysqldump` command with the `people` database (a gentle reminder: the password is `LampIsCool`):

```
$ mysqldump -uapache -p people
Enter password:
```

Notice that this outputs all the SQL needed to create the table and insert all the current records. For more information, see `man mysqldump`.

5.6 SUMMARY

MySQL is a powerful, sophisticated, and easy-to-use SQL database program. Using Perl and DBI, one can easily create programs to automate database management tasks. With this knowledge, the prospective web designer should be able to construct a database-based (for lack of a better term) web site that is portable, sophisticated, easy to manage, and professional appearing. We have examined only a small subset of all that MySQL provides (our 80/20 rule in effect).

5.7 RESOURCES

Books

[DuBois+ 99] DuBois, Paul, and Michael Widenius. *MySQL.* Covers MySQL for the newbie and the experienced user.

[Hatch+ 02] Hatch, Brian, James Lee, and George Kurtz. *Hacking Linux Exposed: Linux Security Secrets and Solutions, Second Edition.* Be sure to read the discussion on how to create good passwords.

[Yarger+ 99] Yarger, Randy Jay, George Reese, and Tim King. *MySQL and mSQL.* An excellent book that covers both MySQL and mSQL, two common databases on Linux systems.

Web Site

MySQL home page: `www.mysql.com/`