CHAPTER 4

Working with Databases

Modern web sites are incredibly powerful, and much of this power derives from their ability to store information. Storing information allows developers to create highly customizable interactions between their software and its users, ranging from entry-based blogs and commenting systems to high-powered banking applications that handle sensitive transactions securely.

In this chapter, I'll cover the basics of MySQL, a powerful, open-source database. Subjects I'll cover include:

- The basics of MySQL data storage
- Manipulating data in MySQL tables
- Your options in PHP for interacting with MySQL databases
- Table structure, usage, and a crash-course in planning

This is the last chapter that covers the theory of building PHP applications; in Chapter 5, I'll begin covering the practical aspects of building a blog!

The Basics of MySQL Data Storage

MySQL is a *relational database management system*, which lets you store data in multiple tables, where each table contains a set of named columns and each row consists of a data entry into the table. Tables will often contain information about other table entries, which allows developers to group relevant information into smaller groups to ease a script's load on the server, as well as simplifying data retrieval.

For example, take a look at how you might store information about musical artists (see Tables 4-1 and 4-2).

Table 4-1. *The* `artists` *Table*

artist_id	artist_name
1	Bon Iver
2	Feist

Table 4-2. The albums Table

album_id	artist_id	album_name
1	1	For Emma, Forever Ago
2	1	Blood Bank - EP3
3	2	Let It Die
4	2	The Reminder

The first table, `artists`, includes two fields. The first field, `artist_id`, stores a unique numerical identifier for the artists. The second column, `artist_name`, stores the artist's name.

The second table, `albums`, stores a unique identifier for each album in the `album_id` column and the album name in the—you guessed it!—`album_name` column. The `album` table includes a third column, `artist_id`, that relates the `artists` and `albums` tables. This column stores the unique artist identifier that corresponds to the artist that recorded the album.

Manipulating Data in MySQL Tables

You can manipulate the data in a MySQL table via several types of MySQL statements. In this section, you will learn the MySQL statements that perform the following actions:

- Create a database
- Create a table in the database
- Insert data into the table
- Retrieve the data from the table
- Modify the data in the table
- Delete the data from the table

You'll test these commands using the phpMyAdmin control panel provided by XAMPP. Open a browser and navigate to `http://localhost/phpmyadmin` to access the control panel (see Figure 4-1).

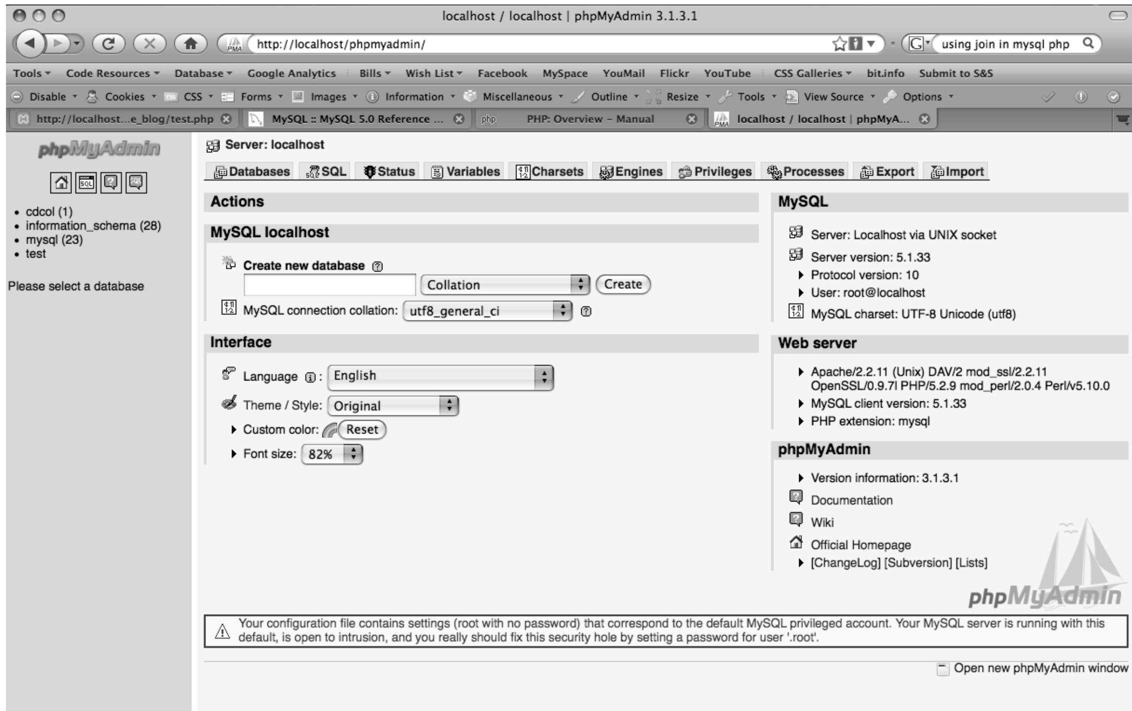

Figure 4-1. *The home page of phpMyAdmin on XAMPP*

Creating and Deleting Databases

The best way to get a feel for creating databases is to create one for testing. By default, XAMPP creates a database called `test`. In the interests of learning, go ahead and delete this database, and then recreate it.

Deleting Databases Using DROP

MySQL uses the word `DROP` to indicate that a database or table should be removed. After you start the `DROP` clause, you need to indicate whether you're removing a database or a table, which you indicate by name, then the name of the database or table (in this case, `test`).

Your complete command should look like this:

```
DROP DATABASE test
```

Click the SQL tab at the top of the screen and enter the preceding command, then click the Go button beneath the SQL field to execute the command and remove the table. Because you're deleting information, an alert pops up asking you to confirm whether you really want to drop the table.

Creating Databases Using CREATE

Next, you need to recreate the test database. MySQL uses the word CREATE to indicate that a table or database is being created. Then, as with the DROP command, you specify a DATABASE and the name of the database: test. Enter this command in the SQL tab, then click the Go button to execute it:

```
CREATE DATABASE test
```

Next, access the test database by clicking its name in the left column of the control panel. Click the SQL tab at the top of the screen, and you're ready to create your first table (see Figure 4-2).

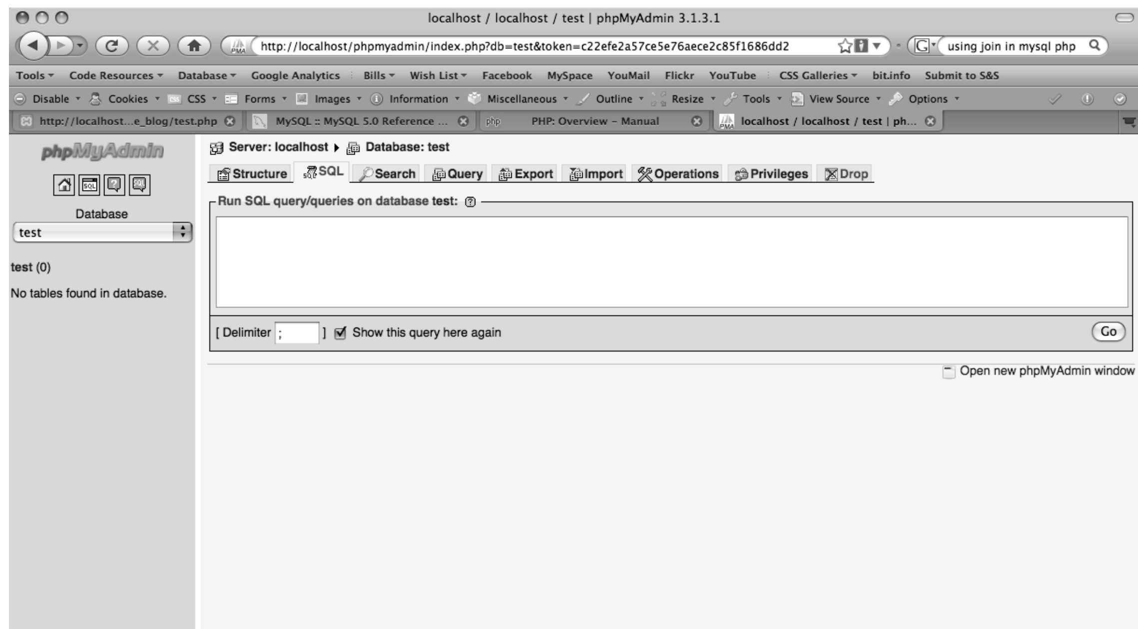

Figure 4-2. *The SQL tab inside the test table on phpMyAdmin*

The CREATE TABLE Statement

Of course, the first thing you need to do to start working with MySQL is to create your first table.

MySQL syntax is very simple because it generally mimics natural speech patterns. In the text field, add the following statement:

```
CREATE TABLE IF NOT EXISTS artists
(
artist_id       INT PRIMARY KEY AUTO_INCREMENT,
artist_name     VARCHAR(150)
)
```

This code creates the `artists` table from the previous example about musicians. The created tables will be empty, but all the columns will be in place.

The statement starts out with the words `CREATE TABLE`, which tells MySQL to do exactly that. You append the optional `IF NOT EXISTS` to ensure you don't overwrite a table if it has already been created, then add the name of the table to be created (`artists`).

The tricky part is defining the columns within the table. To start, you enclose the column names in parentheses after the name of the table, starting with the first column name:

```
(artist_id)
```

To identify the type of information you want to store in this column, you follow the column name with a type identifier:

```
(INT [see Data Types in MySQL])
```

This snippet instructs MySQL to store only integer values in this column.

The `PRIMARY KEY` (see the "Understanding `PRIMARY KEY`" section later in this chapter) and `AUTO_INCREMENT` (see the "Understanding `AUTO_INCREMENT`" section later in this chapter) keywords enable you to make this field update automatically with a unique ID, or *index* (see the "Indexes in MySQL" section later in this chapter).

You create the `artist_name` column with data type `VARCHAR` and specify a maximum length of 150 characters. `VARCHAR` is a variable-length string that can contain anywhere from 0 to 65,535 bytes. You must specify a length with columns of the `VARCHAR` type, or you get an error.

■ **Note** MySQL supports both `VARCHAR` and `CHAR` type columns. The primary difference between the two is that `CHAR` columns are right padded with spaces to fill the specified length, whereas `VARCHAR` columns are not. For example, if the specified field length were 8, the word "data" would be stored as `'data'` in a `VARCHAR` and as `'data    '` in a `CHAR` column.

Clicking the Go button beneath the text field creates the `artists` table. You need to repeat the process and create the `albums` table, which you accomplish using the following code:

```
CREATE TABLE IF NOT EXISTS albums
(
album_id INT PRIMARY KEY auto_increment,
artist_id INT,
album_name VARCHAR(150)
)
```

Again, click the Go button beneath the text field, then click the Structure tab at the top of the field to verify that your tables have been created (see Figure 4-3).

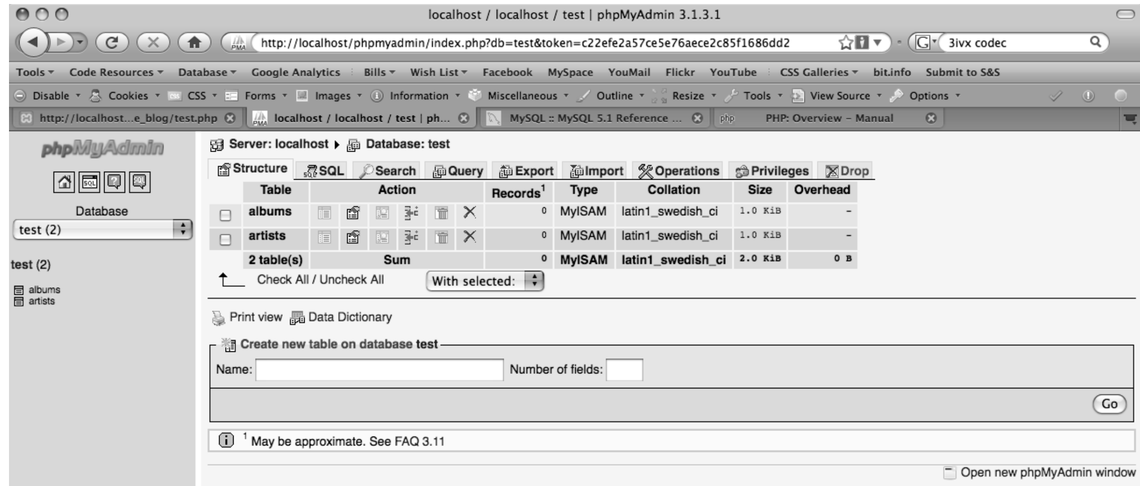

Figure 4-3. *The `albums` and `artists` tables created in phpMyAdmin*

Data Types in MySQL

MySQL supports numerous data types. The types you will use most often include:

- `INT`: an integer value
- `FLOAT`: a floating point number
- `VARCHAR`: a short string value
- `TEXT`: a large string value that is treated as a character string (it's best suited for blog entries)
- `BLOB`: a binary large object that is treated as a binary string (it can store images and similar numbers)
- `DATETIME`: a date value (formatted YYYY-MM-DD HH:MM:SS)

Understanding PRIMARY KEY

A column assigned with the `PRIMARY KEY` identifier should contain a value that uniquely identifies each value throughout the table. Because you're using numerical IDs for each artist, you know that there won't be any overlap in the `artist_id` column.

The use of primary keys in MySQL tables is mandatory because the data isn't really useful if you can't identify entries uniquely.

Understanding AUTO_INCREMENT

Using a unique numerical identifier for individual entries is incredibly useful, so MySQL includes an easy way to create unique identifiers called `AUTO_INCREMENT`. A column flagged with `AUTO_INCREMENT` generates identifiers in sequence automatically as entries are created, starting at `1`.

Indexes in MySQL

When you look up data in a MySQL table, queries can start to back up, depending on the number of rows contained within the table. Imagine a site like CNN.com, where hundreds of thousands of people might be looking up one of the site's articles at any given moment; if each query had to go through every piece of data in every row of the table, sites like this would slow to a crawl under the stress of daily use.

Fortunately, MySQL provides a way to speed up queries by allowing you to create one or more *index* columns, which are sorted snippets of a table's data that enable much faster searching.

In the `artists` table, it makes a lot of sense to use the `artist_id` as an index. Declaring this column as the `PRIMARY KEY` means you've already created an index on it. However, in the `albums` table, the `artist_id` column is not an index, so searching for an artist's recordings could take much longer than you want it to if you had thousands of artists in your database.

To create an index, you'll need to `ALTER` the `albums` table. Next, you add the index by using the `ADD INDEX` clause with the column to index enclosed in parentheses. The complete command looks like this:

```
ALTER TABLE albums ADD INDEX (artist_id)
```

The proper use of MySQL indexes can add a huge performance boost to `SELECT` queries. It should be noted, however, that each index creates a separate column that must be updated every time data is added. This means it can take longer to add data to an indexed column than for an unindexed column.

The INSERT Statement

With your tables created, you're ready to start storing data. The first step is to store artist information in the `artists` table. Each entry must be entered separately, so you start with the first artist, Bon Iver:

```
INSERT INTO artists (artist_name) VALUES ('Bon Iver')
```

The `INSERT INTO` phrase tells MySQL that you're adding information. The next steps are to determine what table you want to add information into (`artists`) and then specify the column(s) you're adding values into (`artist_name`), which you enclose in parentheses.

With your table name and columns selected, you can insert data using the word `VALUES`, followed by the data you wish to insert, enclosed in parentheses:

```
("Bon Iver")
```

You follow the same format to add the next artist:

```
INSERT INTO artists (artist_name) VALUES ('Feist')
```

If you select the artists table from the left-hand column of phpMyAdmin and click the Browse tab, you see that even though you specified only the `artist_name` column, the `artist_id` column was filled out for you automatically (see Figure 4-4).

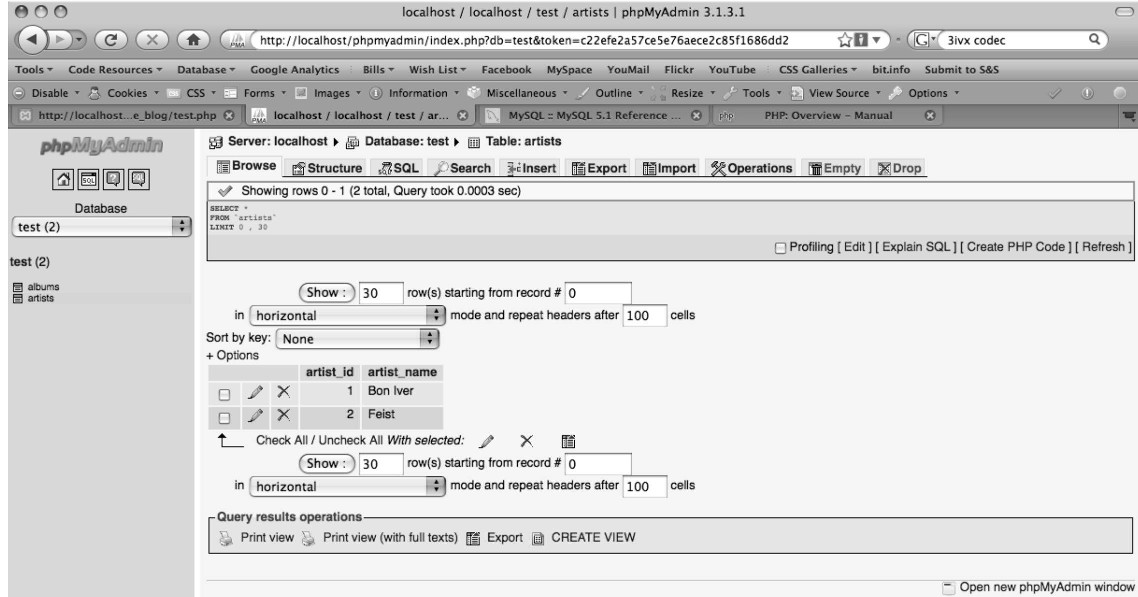

Figure 4-4. *The `artists` table populated with two entries*

To populate the albums table, you specify two columns to enter data into, then execute four statements simultaneously, separating them with a semicolon. Enter the following into the SQL text field and click the Go button:

```
INSERT INTO albums (artist_id, album_name)
VALUES ('1', 'For Emma, Forever Ago'),
('1', 'Blood Bank – EP'),
('2', 'Let It Die'),
('2', 'The Reminder')
```

Instead of executing four different commands to insert the albums, as you did in the preceding example, you can use what is called an *extended insert* to add all four albums at once. This works by enclosing each entry in parentheses, separated by commas.

Now, if you select the albums table from the left column and browse its contents, you see the four entries, as well as the automatically assigned `album_id` values (see Figure 4-5).

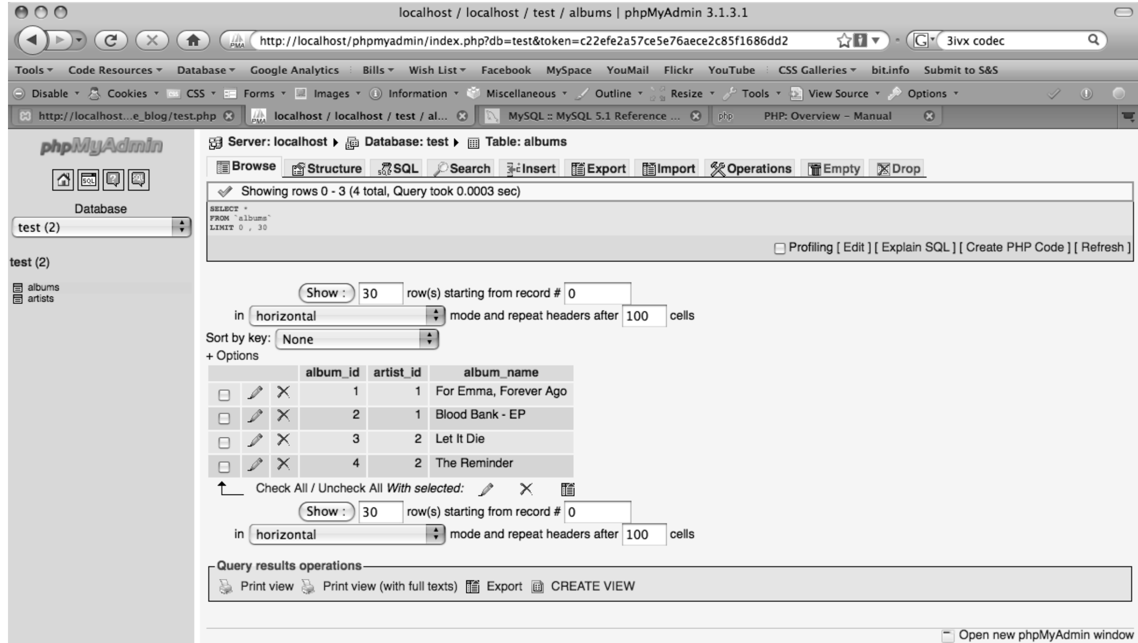

Figure 4-5. *The `albums` table populated with four entries*

The SELECT Statement

Now that you're comfortable inserting data into your tables, you need to figure out how to retrieve it for use with your scripts.

You do this using the `SELECT` statement, followed by the column name(s) you want to retrieve. You specify the table you want to query using the format `FROM table_name`. For example, you can get all album names from the `albums` table by clicking the SQL tab and inserting the following line of code:

```
SELECT album_name FROM albums
```

The result of this query when you execute it in the SQL tab of phpMyAdmin: the four album names (see Figure 4-6).

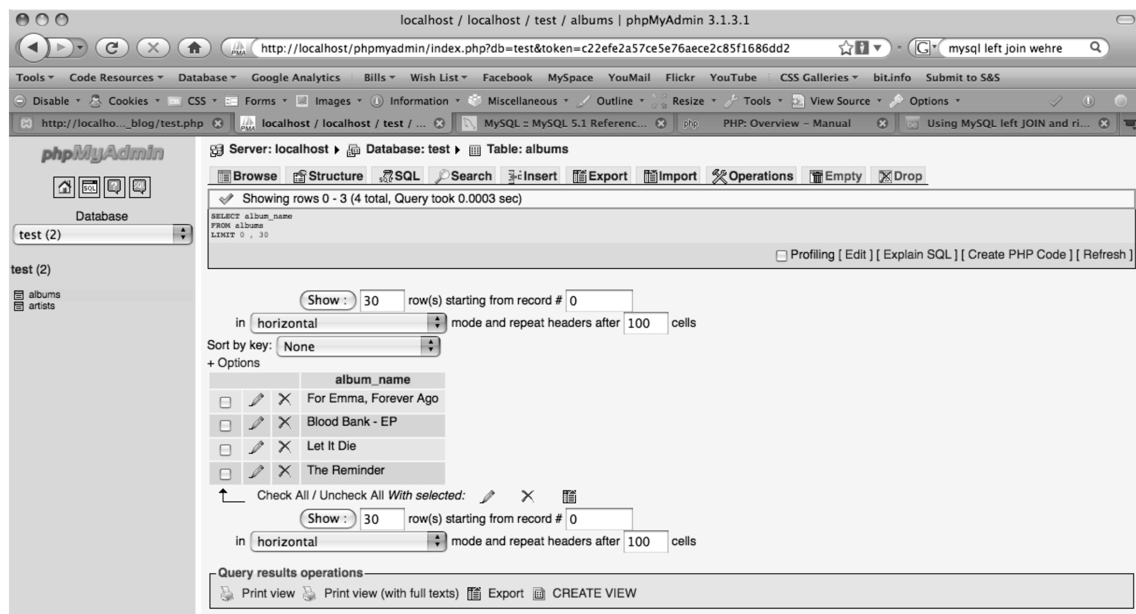

Figure 4-6. *The result of a query for all* `album_name` *column values*

You're also provided with options to modify the results, which enables you to make sure results match certain conditions before they are returned. The `WHERE` clause is the most common query modifier. For example, if you want to retrieve only album titles by Bon Iver, you can use a `WHERE` clause to ensure that returned entries match Bon Iver's `artist_id` of `1`.

```
SELECT album_name FROM albums WHERE artist_id = 1
```

Telling your query to match only the `artist_id` of `1` displays only the albums by Bon Iver (see Figure 4-7).

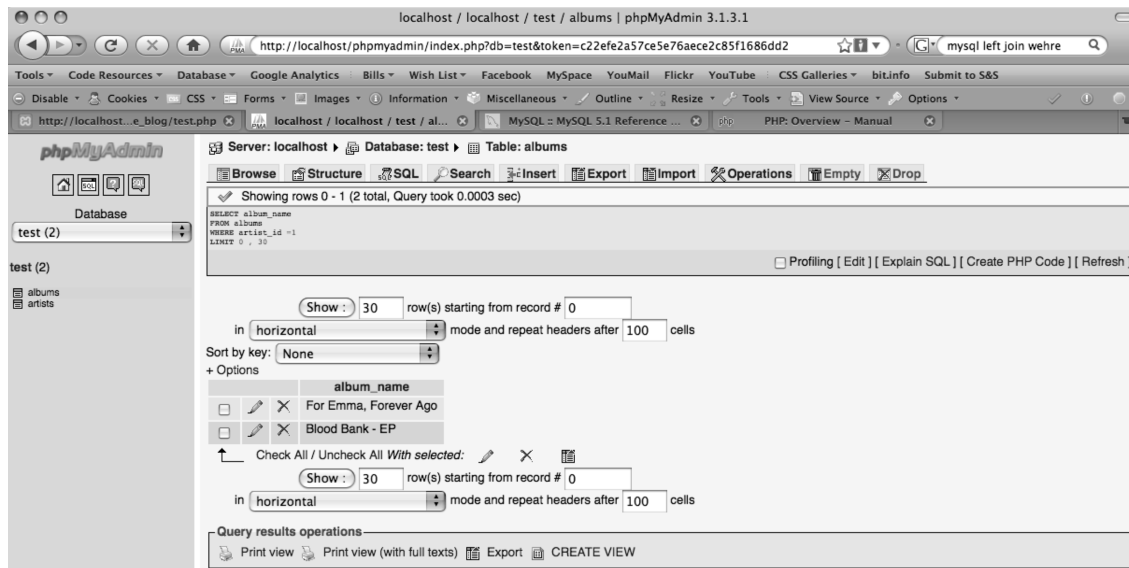

Figure 4-7. *Album names returned from rows that contain the* `artist_id` *of* `1`

■ **Note** There are many other modifiers available you can use in SELECT statements. Refer to the MySQL manual for more information on what they are and how they work. Any further modifiers used throughout this book will be introduced in their appropriate contexts as they come up.

The UPDATE Statement

At some points in your scripts, you will need to change information in an entry. To do this, you use the `UPDATE` statement. For example, if you were to decide to append the release year to Feist's album, "Let It Die," you could do so with the following statement:

```
UPDATE albums
SET album_name = 'Let It Die (2005)'
WHERE album_name = 'Let It Die'
```

You begin by telling MySQL that you're going to update a row using the UPDATE statement and indicating which table name (albums) you want to update. Next, you identify the column(s) to update using the word `SET`, followed by the column name (`album_name`) and the value you wish to update the field with:

```
("Let It Die (2005)")
```

Finally, you add a `WHERE` clause to ensure that only entries that match conditions you set (in this case, `album_name = 'Let It Die'`) are updated.

■Caution When updating rows, it's important to use the `WHERE` clause to limit the rows being updated. Omitting the `WHERE` clause in an update statement will result in all table rows being updated.

The JOIN Statement

Sometimes it's necessary to select information from multiple tables. In the current example, you might have only an artist's ID but need to select both the artist's name and the albums that artist has recorded.

That information is separated between two tables, so one approach is to perform two queries to retrieve the information:

```
SELECT artist_name
FROM artists
WHERE artist_id = 1;
SELECT album_name
FROM albums
WHERE artist_id = 1;
```

Or you can use the `JOIN` statement to select from both tables at once. For example, look closely at you do this first, then I'll go over how it works:

```
SELECT artist_name, album_name
FROM artists
JOIN albums
USING (artist_id)
WHERE artist_id = 1
```

First, you specify that you want to retrieve both the `artist_name` and `album_name` columns, which are stored in the `artists` and `albums` tables, respectively. This might seem wrong at first, because you're selecting from the `artists` table, which would throw an error if not for the `JOIN` clause.

The `JOIN` clause enables you to specify a second table to use in the query (`albums` in this case). This means that the two tables will be combined into one table temporarily, which enables you to retrieve both the `artist_name` and the `album_name` columns with a single query.

However, you must tell MySQL how the two tables are related before this will work. The `artist_id` column exists in both tables, so you're going to use it to tie the two tables together. You accomplish this with the `USING (artist_id)` clause.

Finally, you add a `WHERE` clause, as when using a normal `SELECT` query. In plain English, this preceding query reads, "Retrieve the `artist_name` and `album_name` columns where the `artist_id` field has a value of `1` from the `artists` and `albums` tables."

Executing this query in the SQL tab of `http://localhost/phpmyadmin` returns the following results:

```
Artist_name     album_name
Bon Iver        For Emma, Forever Ago
Bon Iver        Blood Bank - EP
```

The DELETE Statement

If it becomes necessary to remove an entry, you do so using the DELETE statement. For example, assume you want to remove Feist's "The Reminder" from your list altogether:

```
DELETE FROM albums
WHERE album_name = 'The Reminder'
LIMIT 1
```

You start the statement with the `DELETE FROM` statement, then identify the table you want to remove an entry from (`albums`). Next, you create a `WHERE` clause to create the condition you want to match before you delete an entry (`album_name = 'The Reminder'`). And, just to be sure you don't accidentally delete your entire table, you add `LIMIT 1` to the query, which means that only one entry will be deleted, even if more than one entry matches the conditions.

When you click the Go button, phpMyAdmin pops up a confirmation box and asks you if you're sure you want to complete the command. This is your last chance to verify that your command has no errors, such as a missing `WHERE` clause. This confirmation box pops up only when deleting information.

Opening a Connection

You need a method through which your PHP scripts can connect to MySQL in order to interact with the database. You can establish this connection in any of several approaches:

- PHP's MySQL Extension
- PHP's MySQLi Extension
- PHP Data Objects (PDO)

■ **Caution** Due to potential security weaknesses in the MySQL Extension, developers are strongly encouraged to use PDO or MySQLi when using MySQL 4.1.3 or later.

PHP's MySQL Extension

The MySQL Extension is the original extension provided by PHP that allows developers to create PHP applications that interact with MySQL databases earlier than version 4.1.3.

The MySQL Extension uses a *procedural interface*, which means that each action is an individual function (see the code sample that follows). You can use the `artists` table described earlier as a basis for writing a PHP script that retrieve all the artists' names. Open `test.php` in Eclipse and enter the following code:

```
<?php
    // Open a MySQL connection
    $link = mysql_connect('localhost', 'root', '');
    if(!$link) {
        die('Connection failed: ' . mysql_error());
    }
```

```
    // Select the database to work with
    $db = mysql_select_db('test');
    if(!$db) {
        die('Selected database unavailable: ' . mysql_error());
    }

    // Create and execute a MySQL query
    $sql = "SELECT artist_name FROM artists";
    $result = mysql_query($sql);

    // Loop through the returned data and output it
    while($row = mysql_fetch_array($result)) {
        printf("Artist: %s<br />", $row['artist_name']);
    }

    // Free the memory associated with the query
    mysql_free_result($result);

    // Close the connection
    mysql_close($link);
?>
```

Navigating to `http://localhost/simple_blog/test.php` in your browser yields the following result:

```
Artist: Bon Iver
Artist: Feist
```

As the immediately preceding example illustrates, each step in the process has a function assigned to it:

`mysql_connect()`: Accepts the host, username, and password for a MySQL connection. You must call this function before any interaction with the database can take place.

`die()`: This is an alias for the `exit()` command. It stops execution of a script after displaying an optional message (passed as the function argument).

`mysql_error()`: If an error occurs in the MySQL database, this function displays that error; this is helpful for debugging.

`mysql_select_db()`: Selects the database that the script will interact with.

`mysql_query()`: Executes a query to the database. This query can create, modify, return, or delete table rows, as well as perform many other tasks.

`mysql_fetch_array()`: Converts the MySQL resource returned by `mysql_query()` into an array.

`mysql_free_result()`:Frees the memory used by `mysql_query()` to maximize script performance.

`mysql_close()`: Closes the connection opened by `mysql_connect()`.

The MySQL extension doesn't support prepared statements (see the "Using Prepared Statements" section later in this chapter), so it is susceptible to SQL injection, a potentially devastating security issue in web applications. Malicious users can use SQL injection to extract sensitive information from a database, or even go so far as to erase all the information in a given database.

You can minimize this risk by *sanitizing* all information that you want to insert into the database. The MySQL extension provides a function for escaping data called `mysql_real_escape_string()`, which escapes (inserts a backslash before) special characters. Additional functions for sanitizing data, such as `htmlentities()` and `strip_tags()`, are available. However, some risks exist even if you implement these safeguards.

The MySQLi Extension

The MySQL manual recommends that developers using MySQL 4.1.3 or later use the MySQLi extension. There are many benefits to using MySQLi over the original MySQL extension, including MySQLi's:

- Support for both object-oriented and procedural programming methods
- Support for multiple statements
- Enhanced debugging capabilities
- Support for prepared statements

Using Prepared Statements

The MySQLi and PDO extensions provide an extremely useful feature in *prepared statements*.

In a nutshell, prepared statements enable you to separate the data used in a SQL command from the command itself. If you fail to separate these, a malicious user could potentially tamper with your commands. Using a prepared statement means that all submitted data is completely escaped, which eliminates the possibility of SQL injection. You can read more about this subject in Harrison Fisk's article on prepared statements at `http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html`.

A prepared statement works similarly to a regular MySQL statement, except that it uses a placeholder (a question mark [`?`]) to represent data. You can make the best use of prepared statements when use your user input in a query.

For instance, if you have a form on your site that asks what a user's favorite color is, you could use that input in a MySQL query via the `$_POST` superglobal:

```
$sql = "SELECT info FROM colors WHERE color = '$_POST[fav_color]'";
```

However, you aren't performing any sanitization of this input, so a malicious user could potentially exploit your form or harm your site using SQL injection. To avoid this, you can rewrite the preceding statement as a prepared statement:

```
$sql = "SELECT info FROM colors WHERE color = ?";
```

The question mark acts as a placeholder, and it signifies to MySQL that anything passed to this query is to be used only as a parameter for the current statement. This prevents a malicious user from tricking MySQL into giving away information or damaging the database.

Using MySQLi

To use MySQLi, you establish a connection using an object-oriented interface. I'll cover how to take advantage of object-oriented programming (OOP) in the next chapter, as well as discuss the pros and cons of OOP versus procedural programming.

The primary difference between OOP and procedural code is that an object can store information, freeing you from having to pass variables explicitly from function to function.

■ **Note** MySQLi also provides a procedural interface to developers. See the PHP manual entry on MySQLi for more information.

To familiarize yourself with MySQLi, you can rewrite the preceding example using MySQLi. Modify `test.php` so it contains the following:

```
<?php
    // Open a MySQL connection
    $link = new mysqli('localhost', 'root', '', 'test');
    if(!$link) {
        die('Connection failed: ' . $link->error());
    }

    // Create and execute a MySQL query
    $sql = "SELECT artist_name FROM artists";
    $result = $link->query($sql);

    // Loop through the returned data and output it
    while($row = $result->fetch_assoc()) {
        printf("Artist: %s<br />", $row['artist_name']);
    }

    // Free the memory associated with the query
    $result->close();

    // Close the connection
    $link->close();
?>
```

Navigating to `http://localhost/simple_blog/test.php` in your browser yields the following result:

```
Artist: Bon Iver
Artist: Feist
```

MySQLi works similarly to the MySQL extension, with one key exception: instead of providing individual functions, developers using MySQLi have access to *methods*, or functions contained within the MySQLi object. In the preceding code snippet, you instantiate your MySQLi object in the variable `$link` and establish a connection with your host, username, password, and a database name.

To execute a query, you call the `query()` method and pass the variable containing your MySQL statement. You call a method in OOP using the variable that contains the object, followed by an arrow (`->`) and the name of the method you want to call. For example, this line from the previous code example illustrates how to call a method in OOP:

```
$result = $link->query($sql);
```

The `query()` method returns a `mysqli_result` object, which has methods that allow you to access the information returned by the query.

To access each returned entry in order, you set up a loop that uses the result of calling this line:

```
$result->fetch_assoc();
```

Next, you kick out the returned data, then destroy the returned data set by calling the `close()` method on the `$result` object. Also, you close the MySQLi connection by calling the `close()` method on `$link`, as well.

Using Prepared Statements with MySQLi

What really sets MySQLi apart from the MySQL extension is its ability to use prepared statements. If you want to allow a user to select an artist that she wants to see albums from, you can create a form that looks something like this:

```
<form method="post">
    <label for="artist">Select an Artist:</label>
    <select name="artist">
        <option value="1">Bon Iver</option>
        <option value="2">Feist</option>
    </select>
    <input type="submit" />
</form>
```

When the user selects an artist, the artist's unique ID is passed to the processing script in the `$_POST['artist']` variable (review Chapter 3 for a refresher on `$_POST`), which allows you to change your query based on user input.

In `test.php`, you can build a quick script that displays album names based on user input:

```
<?php
    if($_SERVER['REQUEST_METHOD']=='POST')
    {
        // Open a MySQL connection
        $link = new mysqli('localhost', 'root', '', 'test');
```

```
        if(!$link) {
            die('Connection failed: ' . $mysqli->error());
        }

        // Create and execute a MySQL query
        $sql = "SELECT album_name FROM albums WHERE artist_id=?";
        if($stmt = $link->prepare($sql))
        {
            $stmt->bind_param('i', $_POST['artist']);
            $stmt->execute();
            $stmt->bind_result($album);
            while($stmt->fetch()) {
                printf("Album: %s<br />", $album);
            }
            $stmt->close();
        }

        // Close the connection
        $link->close();
    }
    else {
?>

<form method="post">
    <label for="artist">Select an Artist:</label>
    <select name="artist">
        <option value="1">Bon Iver</option>
        <option value="2">Feist</option>
    </select>
    <input type="submit" />
</form>

<?php } // End else ?>
```

When a user submits the form, a new MySQLi object is created, and a query is created with a placeholder for the `artist_id` in the `WHERE` clause. You can then call the `prepare()` method on your MySQLi object (`$link->prepare($sql)`) and pass the query as a parameter.

With your statement (`$stmt`) prepared, you need to tell MySQL how to handle the user input and insert it into the query. This is called *binding parameters* to the query, and you accomplish this by calling the `bind_param()` method on the newly created `$stmt`, which is a `MySQLi_STMT` object.

Binding parameters requires a couple steps: begin by passing the type of the parameter, then pass the parameter value.

MySQLi supports four data types:

- i: Integer (any whole number value)
- s: String (any combination of characters)

- d: Double (any floating point number)
- b: Blob (data is sent in packets that is used for storing images or other binary data)

You're passing the artist's ID, so you set the parameter type to i, then pass the value of $_POST['artist'].

With the parameters bound, you can execute the statement using the execute() method.

After the query is executed, you need to specify variables to contain the returned results, which you accomplish using the bind_result() method. For each column you've requested, you need to provide a variable to contain it. In this example, you need to store the album name, which you accomplish by supplying the $album variable.

Your script now knows where to store returned values, so you can set up a loop to run while results still exist (as returned by the fetch() method). Inside the loop, you output each album name.

Finally, you destroy your resultset and close the connection by calling the close() method on both your MySQLi_STMT and MySQLi objects; this frees the memory used by the query.

If you load your script by navigating to http://localhost/simple_blog/test.php and select Bon Iver from the list, you see the following output:

```
Album: For Emma, Forever Ago
Album: Blood Bank - EP
```

PHP Data Objects (PDO)

PHP Data Objects, or PDO, is similar to MySQLi in that it is an object-oriented approach to handling queries that supports prepared statements.

The main difference between MySQLi and PDO is that PDO is a *database-access abstraction layer*. This means that PDO supports multiple database languages and provides a uniform set of methods for handling most database interactions.

This is a great advantage for applications that need to support multiple database types, such as PostgreSQL, Firebird, or Oracle. Changing from one database type to another generally requires that you rewrite only a small amount of code, which enables developers to change your existing drivers for PDO and continue with business as usual.

The downside to PDO is that some of the advanced features of MySQL are unavailable, such as support for multiple statements. Another potential issue when using PDO is that it relies on the OOP features of PHP5, which means that servers running PHP4 won't be able to run scripts using PDO. This is becoming less of an issue over time because few servers lack access to PHP5; however, it's still something you need to take into consideration when choosing your database access method.

Rewriting Your Example in PDO

You can use PDO to rewrite your prepared statement. In test.php, modify the code as follows:

```
<?php
    if($_SERVER['REQUEST_METHOD']=='POST')
    {
        // Open a MySQL connection
        $dbinfo = 'mysql:host=localhost;dbname=test';
```

```
        $user = 'root';
        $pass = '';
        $link = new PDO($dbinfo, $user, $pass);


        // Create and execute a MySQL query
        $sql = "SELECT album_name
                FROM albums
                WHERE artist_id=?";
        $stmt = $link->prepare($sql);
        if($stmt->execute(array($_POST['artist'])))
        {
            while($row = $stmt->fetch()) {
                printf("Album: %s<br />", $row['album_name']);
            }
            $stmt->closeCursor();
        }
    }
    else {
?>

<form method="post">
    <label for="artist">Select an Artist:</label>
    <select name="artist">
        <option value="1">Bon Iver</option>
        <option value="2">Feist</option>
    </select>
    <input type="submit" />
</form>

<?php } // End else ?>
```

The first step, opening the database connection, is a little different from the other two methods you've learned about so far. This difference stems from the fact that PDO can support multiple database types, which means you need to specify a driver to create the right type of connection.

First, you create a variable called `$dbinfo` that tells PDO to initiate itself using the MySQL driver for the host `localhost` host and the `test` database. Next, you create two more variables, `$user` and `$pass`, to contain your database username and password.

After you pen your connection, you form your query with a placeholder, pass it to the `prepare()` method, and then pass the query to be prepared. This returns a `PDOStatement` object that you save in the `$stmt` variable.

Next, you call the `execute()` method with an array containing the user-supplied artist ID, `$_POST['artist']`. This is equivalent to calling both `bind_param()` and `execute()` with the MySQLi extension.

After the statement has executed, you set up a loop to run while results still exist. Each result is sent to the browser, and you free the memory using the `closeCursor()` method.

Running this script by loading `http://localhost/simple_blog/test.php` produces the following:

```
Album: For Emma, Forever Ago
Album: Blood Bank - EP
```

■ **Note** PDO is highly versatile, so I rely on it for most of this book's examples. Feel free to substitute another method, but be advised that code that interacts with the database will look differently if you do.

Table Structure and a Crash Course in Planning

As your PHP applications become more complicated, your app's performance will start to play a key role in development. MySQL is a potential performance killer in applications, creating bottlenecks that prevent scripts from executing as quickly as you want them to.

Part of your role as a developer is to know the risks involved with MySQL queries and eliminate as many performance issues as possible. The most common risks I'll show you how to address in this chapter include:

- Poor planning of database tables
- Requests for unnecessary data (using the shortcut selector [*])

■ **Note** Database architecture and optimization is a huge task; it's actually a career in and of itself. This book covers only the basics of this subject; however, knowing about the potential performance problems associated with your databases is important because you might not always have a database architect available to you.

Planning Database Tables

Database tables are fairly simple, but interconnected tables can be a hindrance to your scripts' performance if you don't plan them properly. For example, a list of blog entries (`entries`) and a list of a web site's pages (`pages`) might look something like this (see Tables 4-3 and 4-4).

Table 4-3. *The `entries` Table*

title	textSample
Entry	This is some text.
Entry Title	This is more text.
Example Title	A third entry.

Table 4-4. *The pages Table*

page_name	type
Blog	Multi
About	Static
Contact	Form

You might use another table (entry_pages) to link the entries to the page you want to display them on (see Table 4-5).

Table 4-5. *The entry_pages Table*

page	entry
Blog	Sample Entry
Blog	Entry Title
About	Example Title

Unfortunately, you're storing redundant data in this case, which is both unnecessary and potentially harmful to your application's performance. If you know you're on the blog page, you can use two queries to retrieve the entry data:

```
<?php
    // Initiate the PDO object
    $dbinfo = 'mysql:dbname=test;host=localhost';
    $user = 'root';
    $pass = '';
    try {
        $db = new PDO($dbinfo, $user, $pass);
    } catch(PDOException $e) {
        echo 'Connection failed: ', $e->getMessage();
    }

    // Creates the first query
    $sql = "SELECT entry
        FROM entry_pages
        WHERE page='Blog'";
```

```
    // Initialize the $entries variable in case there's no saved data
    $entries = NULL;

    // Retrieves the entries from the table
    foreach($db->query($sql) as $row) {
        $sql2 = "SELECT text
            FROM entries
            WHERE title='$row[entry]'";
        foreach($db->query($sql) as $row2) {
            $entries[] = array($row['title'], $row2['entry']);
        }
    }

    // Display the output
    print_r($entries);
?>
```

This code returns the following:

```
Array
(
    [0] => Array
        (
            [0] => Sample Entry
            [1] => This is some text.
        )

    [1] => Array
        (
            [0] => Entry Title
            [1] => This is more text.
        )
)
```

■ **Note** For the preceding code to work, you need to create the `entry_pages` table and populate it with the data described in Table 4-5.

However, this approach is extremely inefficient because you're retrieving redundant data from the `entry_pages` and `entries` tables.

One way to avoid this problem is to add an additional column to tables containing an ID for each row that will automatically increment as rows are added. This allows for less redundancy in data storage. For example, assume you want to revisit your `pages` and `entries` by adding an ID column (see Tables 4-6 and 4-7).

Table 4-6. The Revised `pages` Table

id	page_name	type
1	Blog	Multi
2	About	Static
3	Contact	Form

Table 4-7. The Revised `entries` Table

id	page_id	title	text
1	1	Sample Entry	This is some text.
2	1	Entry Title	This is more text.
3	2	Example Entry	This is a third entry.

Adding an ID column enables you to eliminate `entry_pages` altogether, which leaves you with an easy way to cross-reference your tables. This means you can rewrite your script to retrieve entries for a particular page far more efficiently:

```
<?php
    $dbinfo = 'mysql:dbname=test;host=localhost';
    $user = 'root';
    $pass = '';

    try {
        $db = new PDO($dbinfo, $user, $pass);
    } catch(PDOException $e) {
        echo 'Connection failed: ', $e->getMessage();
    }

    $sql = "SELECT title, text
            FROM entries
            WHERE page_id=1";
    foreach($db->query($sql) as $row) {
        $entries[] = array($row['title'], $row['text']);
    }

    print_r($entries);
?>
```

This code returns an identical result to your previous example, but takes only half as long to execute because it executes only one query instead of the original two.

■ **Note** Database design is a highly specialized area of programming. You can learn more this subject by picking up a copy of *Beginning Database Design: From Novice to Professional* by Clare Churcher (Apress, 2007).

The Shortcut Selector (*)

MySQL provides a shortcut selector (`SELECT *`) that enables developers to select all data contained within a table. At first glance, this seems like a convenient way to retrieve data from our tables easily. However, the shortcut selector poses a threat to the performance of your scripts by requesting data that isn't needed, which consumes memory unnecessarily.

To avoid wasting resources, it's considered best practice to request all information required by your scripts explicitly. For example, you shouldn't use this approach if you can help it:

```
SELECT * FROM entries
```

Instead, you should write code that goes something like this:

```
SELECT title, text, author FROM entries
```

This ensures that information you won't need for a particular script isn't loaded and thus saves memory. This approach has the added benefit of simplifying your code maintenance because you won't have to remember which columns are returned from table if the code needs to be updated in the future—that information is readily available.

Summary

In this chapter, you've learned the basics of MySQL statements, as well as how to interact with the database from your PHP scripts.

In the next chapter, you'll learn how to begin building your blog by creating a basic entry manager that will allows you to create, modify, and delete entries, as well as display them on a public page.

Recommended Reading

You might find the following links useful for drilling down in more detail on several of the topics covered in this chapter.

- *The MySQL Extension*: `http://us2.php.net/manual/en/book.mysql.php`
- *The MySQLi Extension*: `http://us3.php.net/mysqli`
- *PHP Data Objects (PDO)*: `http://us.php.net/manual/en/book.pdo.php`