Apress™
Books for Professionals by Professionals

Chapter Eleven: "Databases"

A Programmer's Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

CHAPTER 11

Databases

The ability to efficiently store and retrieve large amounts of information has contributed enormously to the success of the Internet. Usually this information storage is implemented through the use of a database (db). Sites such as Yahoo, Amazon, and Ebay depend heavily on the reliability of their databases for storing enormous amounts of information. However, db support is certainly not limited to the behemoth corporations of the Web, as several powerful database implementations are available at relatively low cost (or even free) to Web developers.

When a database is properly implemented, using it to store data results in faster and more flexible retrieval of data. Adding searching and sorting features to your site is greatly simplified, and control over viewing permissions becomes a nonissue by way of the privilege control features in many database systems. Data replication and backup are also simplified.

This chapter begins with an in-depth discussion of how PHP is used to mine and update information in what is arguably PHP's most popular database counterpart, MySQL (http://www.mysql.com). I use MySQL to illustrate how PHP can display and update data in a database, with examples of a basic search engine and sorting mechanism, both of which prove useful in many Web applications. I continue the database discussion with a look into PHP's ODBC (Open Data Base Connectivity) features, which provide a single interface that can be used for simultaneously connecting to several different databases. PHP's ODBC support is demonstrated by an illustration of the process of connecting and retrieving data from a Microsoft Access database. The chapter concludes with a project showing you how to use PHP and a MySQL database to create a categorized online bookmark repository. Users will be able to add information about their favorite sites, placing each site under a predefined set of categories specified by the administrator.

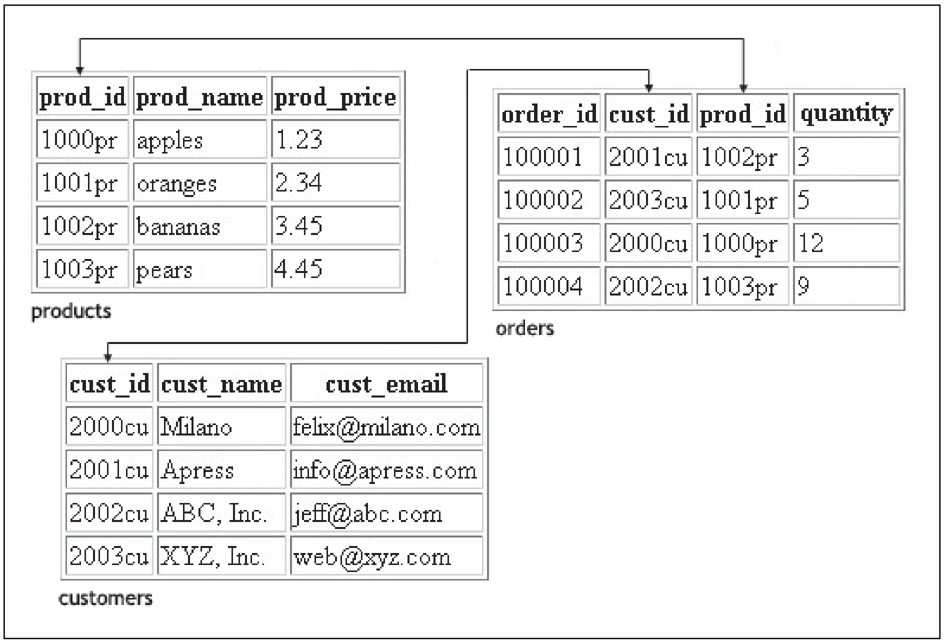
Before commencing the discussion of MySQL, I would like to brief those users unfamiliar with what has become the worldwide standard database language, Structured Query Language (SQL). SQL forms the foundation of practically every popular database implementation; therefore it is imperative that you understand at least the basic underlying ideas of SQL before proceeding to the many database examples throughout the remainder of this book.

What Is SQL?

SQL could be defined as the standard language used to interact with relational databases, discussed shortly. However, SQL is not a computer language like C, C++, or PHP. Instead, it is an interfacing tool for performing various database management tasks, offering a predefined set of commands to the user. Much more than just a query language, as its name implies, SQL offers an array of tools for interacting with a database, including the following:

- **Data structure definition**: SQL can define the various constructs that the database uses to store the data.
- **Data querying**: SQL can retrieve data in the database and present it in an easily readable format.
- **Data manipulation**: SQL can insert, update, and delete database data.
- **Data access control**: SQL makes it possible to coordinate user-specific control over who is capable of viewing, inserting, updating, and deleting data.
- **Data integrity**: SQL prevents data corruption that could be caused by such problems as concurrent updates or system failures.

Note from the SQL definition that SQL's use is specific to *relational* databases. A *relational database* is essentially a database implementation where all data is organized into related table structures. It is possible to create tables that "relate" to one another through the use of data inferences from one table to another. You can think of a *table* as a gridlike arrangement of data values, the position of each determined by a row/column position, which is generally how they're displayed. A sample relational database is illustrated in Figure 11-1.



prod_id	prod_name	prod_price
1000pr	apples	1.23
1001pr	oranges	2.34
1002pr	bananas	3.45
1003pr	pears	4.45

products

order_id	cust_id	prod_id	quantity
100001	2001cu	1002pr	3
100002	2003cu	1001pr	5
100003	2000cu	1000pr	12
100004	2002cu	1003pr	9

orders

cust_id	cust_name	cust_email
2000cu	Milano	felix@milano.com
2001cu	Apress	info@apress.com
2002cu	ABC, Inc.	jeff@abc.com
2003cu	XYZ, Inc.	web@xyz.com

customers

Figure 11-1. A sample relational database (entitled "company")

As you can see in Figure 11-1, each table is arranged in a row/column structure, with each column given a unique title (the scope of the uniqueness limited to the table itself). Notice how a relation is drawn between the customer and orders table, using the cust_id to correctly identify a customer without having to redundantly include the customer's name and other information. There also exists another relation, this one between the orders and products tables. This relation is drawn using the prod_id, which identifies the product that the customer (specified by cust_id) has ordered. Using these relations, you can easily make inferences to both the customer and product information, simply by using these unique identification numbers. As you can now surmise, when used properly relational databases are a powerful tool for organizing and efficiently storing data with a minimum of redundancy. Keep the company database in mind, as I will refer to it frequently in later examples.

So how does SQL communicate with the relational database? This is accomplished through the tailored use of generally defined commands. These commands are clearly descendant from our own spoken language, using easily understood English verbs such as *select, insert, update,* and *delete.* For example, referring to Figure 11-1, if you wanted to retrieve the email of the customer having the identification number 2001cu, you could execute this SQL command:

```
SELECT cust_email FROM customers WHERE cust_id = '2001cu'
```

Logical enough, right? This command could be generalized as follows:

```
SELECT column name FROM table name [ WHERE some condition]
```

The square brackets around the concluding part of the generalized command mean that it is optional. For example, if you wanted to retrieve all of the customer emails from the *customers* table, you could query the database using the following command:

```
SELECT cust_email FROM customers
```

Moving onward, assume that you wanted to insert a new row of data into the products table (thus a new product, since it is assumed that each product is unique). A sample insertion command is:

```
INSERT into products VALUES ('1009pr', 'Red Tomatoes', '1.43');
```

Suppose that you later wanted to delete that data. A sample deletion command is:

```
DELETE FROM products WHERE prod_id = '1009pr';
```

There are many SQL command variations, a complete introduction of them certainly out of the scope of this book. Entire books are devoted to just this subject! However, I will attempt to keep the SQL commands throughout the remainder of this book relatively simple, while at the same time attaining a certain level of practicality in the examples. I suggest searching the Web for several of the many SQL resources and primers. I have included a few of my personal favorites at the conclusion of this section.

NOTE *It is not required that you capitalize the SQL commands in a query. This is my personal preference, done so as to more easily distinguish the query's various components.*

Given the fact that you are reading this book, you are likely already wondering how a database is accessed from the Web environment. Typically, some interfacing language such as PHP, Java, or Perl is used to initiate a connection with the database, and then through the use of predefined functionality the database is queried as necessary. You can think of this interface language as the "glue" that melds the database and the Web together. With that said, I turn my attention to my favorite glue language, PHP.

Additional Resources

Here are a few online resources that the SQL novice and expert alike may find useful.

- Various SQL tutorials:
  `http://perl.about.com/compute/perl/cs/beginningsql/index.htm`

- SQLCourse.com (also offers an onsite practice database):
  `http://www.sqlcourse.com/`

- SQL for Web nerds:
  `http://www.arsdigita.com/books/sql/`

- SQL introduction (focus on MySQL):
  `http://www.devshed.com/Server_Side/MySQL/Intro/`

PHP's Extensive Database Support

If I could name any single most important feature of PHP, it would likely be its database support. PHP offers vast support for practically every prominent db server available today, including those listed below:

Adabas D	Informix	PostgreSQL
Dbase	Ingres	Solid
Direct MS-SQL	InterBase	Sybase
Empress	mSQL	UNIX dbm
FilePro (read-only)	MySQL	Velocis
FrontBase	ODBC	
IBM DB2	Oracle (OCI7 and OC18)	

As you can see from the preceding list, PHP's database support options are extensive, including compatibility with many databases that you have certainly heard of (Oracle, for example) and likely several that you haven't. The bottom line is, if you plan on using a competent database to store your Web information, chances are it will be one that PHP supports. PHP supports a database by offering a set of predefined functions capable of connecting to, querying, and closing the connection to a database.

Discussing the features of each supported database is certainly out of the scope of this book. However, the MySQL database server sufficiently summarizes the general capabilities of many of PHP's supported database servers and serves as a base for any SQL-based server. For this reason, MySQL syntax is used throughout the remainder of this chapter and in any database-related examples in the concluding chapters of this book. Regardless of the database server you decide to implement, you should be able to translate the examples from here on with relative ease.

MySQL

MySQL (`http://www.mysql.com`) is a robust SQL database server developed and maintained by T.c.X DataKonsultAB of Stockholm, Sweden. Publically available since 1995, MySQL has risen to become one of the most popular database servers in the world, this popularity due in part to the server's speed, robustness, and flexible licensing policy. (See note for more information regarding MySQL's licensing strategy.)

Given the merits of MySQL's characteristics, coupled with a vast and extremely easy-to-use set of predefined interfacing functions, MySQL has arguably become PHP's most-popular database counterpart.

NOTE *MySQL is licensed under the GNU General Public License (GPL). Please read the MySQL license information on the MySQL site (http://www.mysql.com) for a full accounting of the current MySQL licensing policy.*

Installation

MySQL is so popular among PHP users that support for the db server is automatically built into the PHP distribution. Therefore, the only task that you are left to deal with is the proper installation of the MySQL package. MySQL is compatible with practically every major operating system, including, among others, FreeBSD, Solaris, UNIX, Linux, and the various Windows versions. While the licensing policy is considerably more flexible than that of other database servers, I strongly suggest taking some time to read through the licensing information found at the MySQL site (http://www.mysql.com).

You can download the latest version of MySQL from one of the many worldwide mirrors. A complete listing of these mirrors is at `http://www.mysql.com/downloads/mirrors.html`. At the time of this writing the latest stable version of MySQL was 3.22.32, with version 3.23 in beta. It is in your best interest to always download the latest stable version. Go to the mirror closest to you and download the version that corresponds with your operating system platform. You'll see links at the top of the page pointing to the most recent versions. Be sure to read through the entire page, as several OS-specific downloads are at the conclusion.

The MySQL development team has done a great job putting together extensive documentation regarding the installation process. I recommend taking some time to thoroughly read through all general installation issues in addition to the information that applies to your operating system.

Configuring MySQL

After a successful installation, it is time to configure the MySQL server. This process largely consists of creating new databases and configuring the MySQL *privilege tables*. The privilege tables control the MySQL database access permissions. Correct configuration of these tables is pivotal to securing your database system, and therefore it is imperative that you fully understand the details of the privilege system before launching your site into a production environment.

Although a chore to learn at first, the MySQL privilege tables are extremely easy to maintain once you understand them. A complete introduction to these tables is certainly out of the scope of this book. However, a number of resources available on the Web are geared toward bringing MySQL users up to speed. Check out the MySQL site (http://www.mysql.com) for further information.

Once you have correctly installed and configured the MySQL distribution, it's time to begin experimenting with Web-based databasing! The next section turns our attention towards exactly this matter, starting with an introduction of PHP's MySQL functionality.

PHP's Predefined MySQL Functions

Once you have created and successfully tested the necessary permissions, you are ready to begin using the MySQL server. In this section, I introduce the predefined MySQL functions, enabling you to easily interface your PHP scripts with a MySQL server. Here is the general order of events that take place during the MySQL server communications process:

1. Establish a connection with the MySQL server. If the connection attempt fails, display an appropriate message and exit process.

2. Select a database on the MySQL server. If you cannot select the database, display an appropriate message and exit process. It's possible to simultaneously have several databases open for querying.

3. Perform necessary queries on selected database(s).

4. Once the querying is complete, close the database server connection.

The example tables (products, customers, orders) in Figure 11-1 are used as the basis for the examples in the remainder of this section. If you would like to follow along with these examples, I suggest going back and creating them now. Alternatively, make a copy of the pages so you do not have to continuously flip back and forth.

With that said, let's begin at the beginning, that is, how to connect to the MySQL database server.

mysql_connect()

The function `mysql_connect()` is used to establish an initial connection with the MySQL server. Once a successful connection is established, a database residing on that server can be selected. The syntax is:

```
int mysql_connect([string hostname [:port] [:/path/to/socket] [, string username]
[, string password])
```

The hostname is the name of the host as listed in the MySQL server privilege tables. Of course, it is also used to direct the request to the Web server hosting the MySQL server, since it is possible to connect to a remote MySQL server. An optional *port* number can be included along with the host, in addition to an optional path to a socket when a local host is specified. Both the *username* and *password* input parameters should correspond to the username and password, respectively, as specified in the MySQL server privilege tables. Note that all of the input parameters are optional, since the privilege tables can be loosely configured to accept a nonauthenticated connection. If the *hostname* parameter is empty, `mysql_connect()` attempts to connect to the local host.

An example connection call follows:

```
@mysql_connect("localhost", "web", "4tf9zzzf") or die("Could not connect to MySQL
server!");
```

In this case, `localhost` is the server host, `web` is the username, and `4tf9zzzf` is the password. The `@` preceding the `mysql_connect()` function will suppress any error message that results from a failed attempt, instead producing the custom one specified in the `die()` call. Note that no value is returned from the `mysql_connect()` call. This is fine when there is only one MySQL server connection that will come into play. However, when connections are made to multiple MySQL servers on multiple hosts, a link ID must be generated so that subsequent commands can be directed to the intended MySQL server. For example:

```
<?
$link1 = @mysql_connect("www.somehost.com", "web", "abcde") or die("Could not
connect to MySQL server!");
$link2 = @mysql_connect("www.someotherhost.com", "usr", "secret") or die("Could
not connect to MySQL server!");
?>
```

Now, `$link1` and `$link2` can be called as needed in subsequent queries. You will soon learn exactly how these link IDs are used in queries to specify the intended server.

NOTE *The function* `mysql_pconnect()` *offers persistent connection support. In multiuser environments,* `mysql_pconnect()` *is recommended over* `mysql_connect()` *as a means for conserving system resources. The* `mysql_pconnect()` *input and return parameters are exactly the same as in* `mysql_connect()`*.*

mysql_select_db()

Once a successful connection is established with the MySQL server, a database residing on that server can be selected. This is accomplished with `mysql_select_db()`. Its syntax is:

```
int mysql_select_db (string database_name [, int link_id])
```

The input parameter `database_name` should be selected and assigned an identification handle (returned by `mysql_select_db()`). Note that the input parameter `link_id` is optional. This is true only when just a single MySQL server connection is open. When multiple connections are open, `link_id` must be specified. An example of how a database is selected using `mysq(_select_db()` follows:

```
<?
@mysql_connect("localhost", "web", "4tf9zzzf")
or die("Could not connect to MySQL server!");
@mysql_select_db("company") or die("Could not select company database!");
?>
```

If there is only one database selection, there is no need to return a database ID. However, as with `mysql_connect()`, when multiple databases are open, the database ID must be returned so there is a way to specify exactly which database you would like to perform a query on; otherwise the most recently *opened* link is used.

mysql_close()

Once you have finished querying the MySQL server, you should close the connection. The function `mysql_close()` will close the connection corresponding to the optional input parameter `link_id`. If the `link_id` input parameter is not specified, `mysql_close()` will close the most recently opened link. The syntax is:

```
int mysql_close ([int link_id])
```

An example of `mysql_close()` follows:

```
<?
@mysql_connect("localhost", "web", "4tf9zzzf")
        or die("Could not connect to MySQL server!");
@mysql_select_db("company") or die("Could not select company database!");
print "You're connected to a MySQL database!";
mysql_close();
?>
```

In the above example, there is no need to specify a link identifier, since only one open server connection exists when `mysql_close()` is called.

NOTE *It is not necessary to close database server connections opened by* `mysql_pconnect()`*.*

mysql_query()

The function `mysql_query()` provides the functional interface from which a database can be queried. Its syntax is:

```
int mysql_query (string query [, int link_id])
```

The input parameter query corresponds to an SQL query. This query is sent either to the server connection corresponding to the last opened link or to the connection specified by the optional input parameter `link_id`.

People often mistakenly think that the `mysql_query()` function returns the results of the query. This is not the case. Depending on the type of query, `mysql_query()` has different outcomes. In a successful SELECT SQL statement, a result ID is returned that can subsequently be passed to `mysql_result()` so the selected data can be formatted and displayed to the screen. If the query fails, FALSE is returned. The function `mysql_result()` is introduced later in this section. Furthermore, the number of rows that have been selected can be determined by executing `mysql_num_rows()`. This function is also introduced later in this section.

In the case of SQL statements involving INSERT, UPDATE, REPLACE, or DELETE, the function `mysql_affected_rows()` can be called to determine how many rows were affected by the query. (The function `mysql_affected_rows()` is introduced next.)

With that said, I will delay presenting an example until the `mysql_result()` and `mysql_affected_rows()` functions are introduced.

TIP *If you are concerned that you are using up too much memory when making various query calls, call the predefined PHP function* `mysql_free_result()`*. This function, which takes as input the* `result_id` *returned from* `mysql_query()`*, will free up all memory associated with that query call.*

mysql_affected_rows()

It is often useful to return the number of rows affected by an SQL query involving an INSERT, UPDATE, REPLACE, or DELETE. This is accomplished with the function `mysql_affected_rows()`. Its syntax is:

```
int mysql_affected_rows ([int link_id])
```

Notice that the input parameter `link_id` is optional. If it is not included, `mysql_affected_rows()` attempts to use the last opened `link_id`. Consider the following example:

```
<?
// connect to the server and select a database.
@mysql_connect("localhost", "web", "4tf9zzzf")
        or die("Could not connect to MySQL server!");
@mysql_select_db("company") or die("Could not select company database!");
// declare query
$query = "UPDATE products SET prod_name = \"cantaloupe\"
          WHERE prod_id = \"1001pr\"";
// execute query
$result = mysql_query($query);
// determine the number of rows that have been affected.
print "Total row updated: ".mysql_affected_rows();
mysql_close();
?>
```

Executing this code example returns this:

```
Total rows updated: 1
```

This will not work for queries involving a SELECT statement. To determine the number of rows returned from a SELECT, use the function `mysql_num_rows()` instead. This function is introduced next.

CAUTION *There seems to be a quirk when using* `mysql_affected_rows()` *in one particular situation. If you execute a DELETE without a WHERE clause,* `mysql_affected_rows()` *will return 0.*

mysql_num_rows()

The function `mysql_num_rows()` is used to determine the number of rows returned from a SELECT query statement. Its syntax is:

```
int mysql_num_rows (int result)
```

A usage example of `mysql_num_rows()` follows:

```
<?
@mysql_connect("localhost", "web", "4tf9zzzf")
        or die("Could not connect to MySQL server!");
@mysql_select_db("company") or die("Could not select company database!");
// select all product names where the product name begins with a 'p'
$query = "SELECT prod_name FROM products WHERE prod_name LIKE \"p%\"";
// execute the query
$result = mysql_query($query);

print "Total rows selected: ".mysql_num_rows($result);

mysql_close();
?>
```

Since there is only one product name beginning with *p* (pears), only one row is selected. This is the result:

```
Total rows selected: 1
```

mysql_result()

The function `mysql_result()` is used in conjunction with `mysql_query()` (when a SELECT query is involved) to produce a data set. Its syntax is:

```
int mysql_result (int result_id, int row [, mixed field])
```

The input parameter `result_id` refers to a value returned by `mysql_query()`. The parameter `row` refers to a particular row in the dataset specified by the `result_id`. Lastly, the optional input parameter `field` can be used to specify the following:

- Field offset in the table.
- Field name
- Field name specified in dot format. Dot format is simply another way to specify the field name, specified as `fieldname.tablename`.

Consider Listing 11-1, which makes use of the database displayed in Figure 11-1.

Listing 11-1: Retrieving and formatting data in a MySQL database

```
<?
@mysql_connect("localhost", "web", "ffttss") or die("Could not connect to MySQL
server!");
@mysql_select_db("company") or die("Could not select company database!");
// Select all rows in the products table
$query = "SELECT * FROM products";
$result = mysql_query($query);
$x = 0;
print "<table>";
print "<tr><th>Product ID</th><th>Product Name</th><th>Product Price</th></tr>";
while ($x < mysql_numrows($result)) :
     $id = mysql_result($result, $x, 'prod_id');
     $name = mysql_result($result, $x, 'prod_name');
     $price = mysql_result($result, $x, 'prod_price');
     print "<tr>";
     print "<td>$id</td><td>$name</td><td>$price</td>";
     print "</tr>";
     $x++;
endwhile;
</table>
mysql_close();
?>
```

Executing this example using our sample data returns the results you see in Listing 11-2:

Listing 11-2: Output generated from execution of Listing 11-1

```
<table>
<tr>
<th>Product ID</th><th>Product Name</th><th>Product Price</th>
</tr>
<tr>
<td>1000pr</td>
<td>apples</td>
<td>1.23</td>
</tr>
<tr>
<td>1001pr</td>
<td>oranges</td>
<td>2.34</td>
</tr>
<tr>
```

```
<td>1002pr</td>
<td>bananas</td>
<td>3.45</td>
</tr>
<tr>
<td>1003pr</td>
<td>pears</td>
<td>4.45</td>
</tr>
</table>
```

While this function works fine when dealing with relatively small result sets, there are other functions that operate much more efficiently, namely, `mysql_fetch_row()` and `mysql_fetch_array()`. These functions are described below.

mysql_fetch_row()

It is typically much more convenient to simultaneously assign an entire row to an indexed array (starting at offset 0), rather than make multiple calls to `mysql_result()` to assign column values. This is accomplished with `mysql_fetch_row()` Its syntax is:

```
array mysql_fetch_row() (int result)
```

Using the array function `list()` in conjunction with `mysql_fetch_row()` can eliminate several lines of code necessary when using `mysql_result()`. In Listing 11-3, I reconsider the code used in Listing 11-1, this time using `list()` and `mysql_fetch_row()`.

Listing 11-3: Retrieving data with mysql_fetch_row()

```
<?
@mysql_connect("localhost", "web", "ffttss") or die("Could not connect to MySQL
server!");
@mysql_select_db("company") or die("Could not select company database!");
$query = "SELECT * FROM products";
$result = mysql_query($query);
print "<table>\n";
print "<tr>\n<th>Product ID</th><th>Product Name</th><th>Product
Price</th>\n</tr>\n";
```

```
while (list($id, $name, $price) =  mysql_fetch_row($result)) :
   print "<tr>\n";
   print "<td>$id</td>\n<td>$name</td>\n<td>$price</td>\n";
   print "</tr>\n";
endwhile;
print "</table>";
mysql_close();
?>
```

Execution of Listing 11-3 will produce the same results as Listing 11-1. However, Listing 11-3 accomplishes this with fewer lines of code.

mysql_fetch_array()

The function `mysql_fetch_array()` accomplishes the same result as `mysql_fetch_row()`, except that by default it assigns a returned row to an *associative* array. However, you can specify the type of array mapping (associative, numerically indexed, or both). The syntax is:

```
array mysql_fetch_array (int result [, result_type])
```

The input parameter `result` is the result returned by a call to `mysql_query()`. The optional input parameter `result_type` can be one of three values:

- `MYSQL_ASSOC` directs `mysql_fetch_array()` to return an associative array. This is the default should `result_type` not be specified.
- `MYSQL_NUM` directs `mysql_fetch_array()` to return a numerically indexed array.
- `MYSQL_BOTH` directs `mysql_fetch_array()` to allow for the returned row to be accessed either numerically or associatively.

Listing 11-4 is a variation of Listing 11-1 and Listing 11-3, this time using `mysql_fetch_array()` to return an associative array of row values.

Listing 11-4: Retrieving data with mysql_fetch_array()

```
<?
@mysql_connect("localhost", "web", "ffttss")
        or die("Could not connect to MySQL server!");
@mysql_select_db("company") or die("Could not select products database!");
$query = "SELECT * FROM products";
```

```
$result = mysql_query($query);

print "<table>\n";
print "<tr>\n<th>Product ID</th><th>Product Name</th>
             <th>Product Price</th>\n</tr>\n";
// No result type, therefore It defaults to MYSQL_ASSOC
while ($row = mysql_fetch_array($result)) :
   print "<tr>\n";
   print "<td>".$row["prod_id"]."</td>\n
          <td>".$row["prod_name"]."</td>\n
          <td>".$row["prod_price"]."</td>\n";
   print "</tr>\n";
endwhile;
print "</table>";
mysql_close();
?>
```

Executing Listing 11-4 yields the same results as Listings 11-1 and 11-3.

At this point, you have been introduced to enough of PHP's MySQL functionality to begin building interesting database applications. The first application that I will consider is a basic search engine. This example will illustrate how HTML forms (introduced in the preceding chapter) are used to supply information that is subsequently used to mine information from a database.

Building a Search Engine

While all of us are certainly familiar with using a Web-based search engine to retrieve data, how is one built? A simple search engine must be able to accept at least one keyword, which is then passed to a SQL query, which in turn polls the database for matches. There are many ways that a search engine could format results (for example, by category or match consistency).

The search engine illustrated in Listing 11-5 is actually geared toward mining for customer information. The search form prompts the user for a keyword and a category (customer name, customer ID, or customer email) from which the search will take place. If the user enters an existing customer name, ID, or email, the engine will query the database for the remaining pieces of information. Then it makes use of the customer ID to poll the orders table for an order history based on that customer. All orders placed by that customer are displayed in descending order. If the input keyword is not in the category chosen by the user, then the search will cease, and the user is provided with an appropriate message, and the form is again displayed.

Listing 11-5: A simple search engine (searchengine.php)

```
<?
$form =
"<form action=\"searchengine.php\" method=\"post\">
<input type=\"hidden\" name=\"seenform\" value=\"y\">
Keyword:<br>
<input type=\"text\" name=\"keyword\" size=\"20\" maxlength=\"20\" value=\"\"><br>
Search Focus:<br>
<select name=\"category\">
<option value=\"\">Choose a category:
<option value=\"cust_id\">Customer ID
<option value=\"cust_name\">Customer Name
<option value=\"cust_email\">Customer Email
</select><br>
<input type=\"submit\" value=\"search\">
</form>
";
// If the form has not been displayed, show it.
if ($seenform != "y") :
     print $form;
else :
     // connect to MySQL server and select database
     @mysql_connect("localhost", "web", "ffttss")
             or die("Could not connect to MySQL server!");
     @mysql_select_db("company") or die("Could not select company database!");
     // form and execute query statement
     $query = "SELECT cust_id, cust_name, cust_email
               FROM customers WHERE $category = '$keyword'";
     $result = mysql_query($query);
     // If no matches found, display message and redisplay form
     if (mysql_num_rows($result) == 0) :
          print "Sorry, but no matches were found. Please try your search again:";
          print $form;
     // matches found, therefore format and display results
     else :
          // format and display returned row values.
          list($id, $name, $email) = mysql_fetch_row($result);
          print "<h3>Customer Information:</h3>";
          print "<b>Name:</b> $name <br>";
          print "<b>Identification #:</b> $id <br>";
          print "<b>Email:</b> <a href=\"mailto:$email\">$email</a> <br>";

          print "<h3>Order History:</h3>";
          // form and execute 'orders' query
          $query = "SELECT order_id, prod_id, quantity
                    FROM orders WHERE cust_id = '$id'
                    ORDER BY quantity DESC";
          $result = mysql_query($query);

          print "<table border = 1>";
          print "<tr><th>Order ID</th><th>Product ID</th><th>Quantity</th></tr>";
          // format and display returned row values.
          while (list($order_id,$prod_id,$quantity) = mysql_fetch_row($result)):
               print "<tr>";
               print "<td>$order_id</td><td>$prod_id</td><td>$quantity</td>";
               print "</tr>";
          endwhile;
          print "</table>";
     endif;
endif;
?>
```

Entering the keyword Milano and selecting Customer Name from the pull-down menu, will cause the following information to be displayed on the screen:

Customer Information:

Name: Milano
Identification #: 2000cu
Email: `felix@milano.com`

Order History:

Order ID	Product ID	Quantity
100003	1000pr	12
100005	1002pr	11

Of course, this is just one of many ways that a search engine could be implemented. Consider adding the possibility to allow multiple keywords, partial keywords, or automated suggestions for keywords not in the table, but with similar matches. I'll leave these features to your creativeness as a programming exercise.

Building a Table Sorter

It is particularly useful for users to be able to sort data as they wish when displaying database data. For example, consider the output shown from the search engine example, in particular the data following the Order History: header. What if the list was particularly long, and you wanted to reorder the data by the product ID? Or by order ID? To illustrate this concept, take a moment to check out one of my favorite sites, http://download.cnet.com. When viewing a particular software category, notice that when you click each header (Title, Date Added, Downloads, and File Size), the list is resorted accordingly. The following code shows just how a feature such as this can be constructed.

In Listing 11-6, I select the data from the orders table. By default, the data is ordered by descending quantity. However, clicking any of the table headers will cause the script to again be called, but this time reordering the table information in accordance with the column in which the user clicked.

Listing 11-6: A table sorter (tablesorter.php)

```
<?
// connect to MySQL server and select database
@mysql_connect("localhost", "web", "ffttss")
        or die("Could not connect to MySQL server!");
@mysql_select_db("company") or die("Could not select company database!");
// If the $key variable is not set, default to 'quantity'
if (! isset($key)) :
     $key = "quantity";
endif;
// create and execute query. Any retrieved data is sorted in descending order
$query = "SELECT order_id, cust_id, prod_id, quantity
          FROM orders ORDER BY $key DESC";
$result = mysql_query($query);
// create table header
print "<table border = 1>";
print "<tr>
<th><a href=\"tablesorter.php?key=order_id\">Order ID</a></th>
<th><a href=\"tablesorter.php?key=cust_id\">Customer ID</a></th>
<th><a href=\"tablesorter.php?key=prod_id\">Product ID</a></th>
<th><a href=\"tablesorter.php?key=quantity\">Quantity</a></th></tr>";
// format and display each row value
while (list($order_id, $cust_id, $prod_id, $quantity) =
mysql_fetch_row($result)) :
     print "<tr>";
     print "<td>$order_id</td><td>$cust_id</td>
            <td>$prod_id</td><td>$quantity</td>";
     print "</tr>";
endwhile;
// close table
print "</table>";
?>
```

Using the information retrieved from the company database (Figure 11-1), the default table displayed by Listing 11-6 follows.

Order ID	Customer ID	Product ID	Quantity
100003	2000cu	1000pr	12
100005	2000cu	1002pr	11
100004	2002cu	1000pr	9
100002	2003cu	1001pr	5
100001	2001cu	1002pr	3

Notice that there are four links shown as table headers. Since Quantity is designated as the default sorting attribute, the rows are sorted in descending order according to quantity. If you click the Order_ID link, you will see that the page reloads, but this time the rows are sorted in accordance with descending order IDs. Thus, the following table would be shown:

Order ID	Customer ID	Product ID	Quantity
100005	2000cu	1002pr	11
100004	2002cu	1000pr	9
100003	2000cu	1000pr	12
100002	2003cu	1001pr	5
100001	2001cu	1002pr	3

As you can see, this feature can prove immensely useful for formatting database information. Just by changing the SELECT clause, you can perform any number of ordering arrangements, including ascending, descending, and grouping information.

And thus finishes the introduction to MySQL. Keep in mind that there is still quite a bit more to be learned about MySQL. For a complete listing of PHP's supported MySQL commands, check out the manual at http://www.php.net/manual.

ODBC

Using a database-specific set of commands is fine when you are sure that you only need to interface with one specific type of database. However, what happens when you need to connect with MySQL, Microsoft SQL Server, and IBM DB2, all in the same application? The same problem arises when you want to develop database-independent applications that can be layered on top of a potential client's existing database infrastructure. ODBC, an acronym for Open Database Connectivity, is an API (application programming interface) used to abstract the database interface calls, resulting in the ability to implement a single set of commands to interact with several different types of databases. The advantageous implications of this should be obvious, since it eliminates the need for you to rewrite the same code repeatedly just to be able to interact with different database brands.

For ODBC to be used in conjunction with a particular database server, that server must be ODBC compliant. In other words, ODBC drivers must be available for it. Check the database's documentation for further information if you are unsure. Once you locate these drivers, you then need to download and install them. Although ODBC, originally created by Microsoft and now an open standard, is predominantly used to access databases developed for the Windows platform, ODBC drivers are also available for the Linux platform. The following links point to some of the more popular drivers available:

- Windows 95/98/NT database drivers (http://www.microsoft.com/data/odbc/)
- Automation Technologies (http://www.odbcsdk.com)
- Easysoft (http://www.easysoft.com/products/oob/main.phtml)
- MySQL's MyODBC drivers (http://www.mysql.com)
- OpenLinkSoftware (http://www.openlinksw.com)

Each ODBC application may vary slightly in usage, platform, and purpose. I would advise reading through the documentation of each to gain a better understanding of the various issues involved with ODBC and these third-party applications. Regardless of their differences, all are known to work well with PHP.

Once you've determined the ODBC application that best fits your purposes, download it and follow the installation and configuration instructions. Then it's time to move on to the next section, "PHP's ODBC Support."

PHP's ODBC Support

PHP's ODBC support, collectively known as the *Unified ODBC Functions*, provide the typical ODBC support in addition to the ability to use these functions to access certain databases that have based their own API on the already existing ODBC API. These database servers are listed in below:

- Adabas D
- IODBC
- IBM DB2
- Solid
- Sybase SQL Anywhere

It is important to note that when using any of these databases, there is actually no ODBC interaction involved. Rather, PHP's Unified ODBC Functions can be used to interface with the database. This is advantageous in the sense that should you choose to use any other ODBC database (or other database listed above), you already have the necessary scripts at your disposal.

NOTE *PHP's ODBC support is built in to the PHP distribution, so there is no need for special configuration options unless otherwise stated.*

There are currently almost 40 predefined Unified ODBC Functions. However, you only need to know a few to begin extracting information from an ODBC-enabled database. I will introduce these necessary functions presently. If you would like a complete listing of all of PHP's predefined ODBC functions, please refer to the PHP manual (http://www.php.net/manual).

odbc_connect()

Before querying an ODBC-enabled database, you must first establish a connection. This is accomplished with `odbc_connect()`. Its syntax is:

```
int odbc_connect (string data_source, string username, string password [,int
cursor_type])
```

The input parameter `data_source` specifies the ODBC-enabled database to which you are attempting to connect. The parameters `username` and `password` specify, logically enough, the username and password required to connect to the `data_source`. The optional input parameter `cursor_type` is used to resolve quirks among some ODBC drivers. There are four possible values for the optional parameter `cursor_type`:

- SQL_CUR_USE_IF_NEEDED
- SQL_CUR_USE_ODBC
- SQL_CUR_USE_DRIVER
- SQL_CUR_DEFAULT

These cursor types attempt to resolve certain errors that arise from use of ODBC drivers. Chances are you won't need to use them, but keep them in mind in case you experience problems when attempting to execute certain queries that your ODBC distribution may not be able to handle.

Implementing `odbc_connect()` is easy. Here is an example:

```
<?
odbc_connect("myAccessDB", "user", "secret")
      or die("Could not connect to ODBC database");
?>
```

TIP *The function* `odbc_pconnect()` *is used to open a persistent database connection. This can save system resources, because* `odbc_pconnect()` *first checks for an already open connection before opening another. If a connection is already open, that connection is used.*

odbc_close()

After you have finished using the ODBC database, you should close the connection to free up any resources being used by the open connection. This is accomplished with `odbc_close()`. Its syntax is:

```
void odbc_close (int connection_id)
```

The input parameter `connection_id` refers to the open connection identifier. Here is a short example:

```
<?
$connect = @odbc_connect("myAccessDB", "user", "secret")
                  or die("Could not connect to ODBC database!");
print "Currently connected to ODBC database!";
odbc_close($connect);
?>
```

odbc_prepare()

Before a query is executed, the query must be "prepared." This is accomplished with `odbc_prepare()`. Its syntax is:

```
int odbc_prepare (int connection_ID, string query)
```

The input parameter `connection_ID` refers to the connection identification variable returned by `odbc_connect()`. The parameter `query` refers to the query that is to be executed by the database server. If the query is invalid and therefore cannot be executed, FALSE is returned; Otherwise, a result identifier is returned that can subsequently be used with `odbc_execute()` (introduced next).

odbc_execute()

After the query is prepared by `odbc_prepare()`, it can then be executed with `odbc_execute()`. Its syntax is:

```
int odbc_execute (int result_ID [, array parameters])
```

The input parameter `result_ID` is a result identifier returned from a successful execution of `odbc_prepare()`. The optional parameter `parameters` only needs to be used if you are passing serializable data into the function.

Consider the following example:

```
<?
$connect = @odbc_connect("myAccessDB", "user", "secret")
                  or die("Could not connect to ODBC database");
$query = "UPDATE customers set cust_id = \"Milano, Inc.\"
        WHERE cust_id \"2000cu\"";
$result = odbc_prepare($connect, $query) or die("Couldn't prepare query!");
$result = odbc_execute($result) or die("Couldn't execute query!");
odbc_close($connect);
?>
```

This example illustrates a complete ODBC transaction when the query does not result in the need to display data to the browser (as would likely be the case with a SELECT statement). A complete ODBC transaction using a SELECT query is shown later in this chapter, under "`odbc_result_all()`".

odbc_exec()

The function `odbc_exec()` accomplishes the roles of both `odbc_prepare()` and `odbc_execute()`. Its syntax is:

```
int odbc_exec (int connection_ID, string query)
```

The input parameter `connection_ID` refers to the connection identification variable returned by `odbc_connect()`. The parameter `query` refers to the query to be executed by the database server. If the query fails, FALSE is returned; otherwise a result identifier is returned, which can be then used in subsequent functions.

```
<?
$connect = @odbc_connect("myAccessDB", "user", "secret")
                  or die("Could not connect to ODBC database!");
$query = "SELECT * FROM customers";
$result = odbc_exec($connect, $query) or die("Couldn't execute query!");
odbc_close($connect);
?>
```

In the above example, `odbc_exec()` will attempt to execute the query specified by `$query`. If it is successfully executed, `$result` is assigned a result identifier. Otherwise, `$result` is assigned FALSE, and the string enclosed in the `die()` function is displayed.

odbc_result_all()

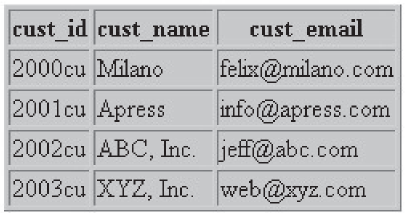
This very cool function will format and display all rows returned by a result identifier produced by `odbc_exec()` or `odbc_execute()`. Its syntax is:

```
int odbc_result_all (int result_ID [, string table_format])
```

The input parameter `result_ID` is a result identifier returned by `odbc_exec()` or `odbc_execute()`. The optional parameter `table_format` takes as input HTML table characteristics. Consider this example:

```
<?
$connect = @odbc_connect("myAccessDB", "user", "secret")
                  or die("Could not connect to ODBC database!");
$query = "SELECT * FROM customers";
$result = odbc_exec($connect, $query) or die("Couldn't execute query!");
odbc_result_all($result, "BGCOLOR='#c0c0c0' border='1'");
odbc_close($connect);
?>
```

Execution of this example would produce a table characterized by a light gray background and a border size of 1 containing the contents of the customers table. Assuming that this table contained the data shown back in Figure 11-1, the table would be displayed as seen in Figure 11-2.



cust_id	cust_name	cust_email
2000cu	Milano	felix@milano.com
2001cu	Apress	info@apress.com
2002cu	ABC, Inc.	jeff@abc.com
2003cu	XYZ, Inc.	web@xyz.com

Figure 11-2. ODBC data as displayed to the browser

odbc_free_result()

It is generally good programming practice to restore any resources consumed by operations that have been terminated. When working with ODBC queries, this is accomplished with `odbc_free_result()`. Its syntax is:

```
int odbc_free_result (int result_id)
```

The input result_id refers to the result identifier that will not be used anymore. Keep in mind that all memory resources are automatically restored when the script finishes; therefore it is only necessary to use odbc_free_result() when particularly large queries are involved that could consume significant amounts of memory. The following example illustrates the syntax of odbc_free_result(). Remember that this function isn't really necessary unless you plan on making several queries throughout a single script, since all memory is returned anyway at the conclusion of the script.

```
<?
$connect = @odbc_connect("myAccessDB", "user", "secret")
                  or die("Could not connect to ODBC database!");
$query = "SELECT * FROM customers";
$result = odbc_exec($connect, $query) or die("Couldn't execute query!");
odbc_result_all($result, "BGCOLOR='#c0c0c0' border='1'")
odbc_free_result($result);
odbc_close($connect);
?>
```

After odbc_result_all() has finished using the result identifier, the memory is recuperated using odbc_free_result().

This concludes the summarization of those PHP ODBC functions that are particularly indispensable when creating simple ODBC interfaces through the Web In the next section, "Microsoft Access and PHP," many of these functions are used to illustrate just how easily PHP can be used to interface with one of the more popular database servers, Microsoft Access.

Microsoft Access and PHP

Microsoft's Access Database (http://www.microsoft.com/office/access/) is a popular database solution due in large part to its user-friendly graphical interface. It alone can be used as the database solution, or its graphical interface can be used as a front end to interface with other databases, such as MySQL or Microsoft's SQL Server.

To illustrate the use of PHP's ODBC support, I'll describe how you can connect to an MS Access database using PHP. It is surprisingly easy and is a great addition to your PHP programming repertoire, due to the popularity of Microsoft Access. I'll detail this process step by step:

1. Go ahead and create an Access database. I'll assume that you know how to do this already. If you don't, but still want to follow along with this example, just quickly create a database using the Access Wizard. For this example, I created the predefined Contact Management Database using

the wizard. Be sure to insert some information into a table and take note of that table name, as you will need it in a while!

2. Save the database somewhere on your computer.

3. Now it's time to make that Access database available using ODBC. Go to Start -> Settings -> Control Panel. You should see an icon entitled "ODBC Data Sources (32 bit)." This is the ODBC Administrator, used to handle the various drivers and database sources on the system. Open this icon by double-clicking it. The window will open to the default tabbed sub-window entitled "User DSN (User Data Sources)." The User DSN contains the data sources specific to a single user and can be used only on this machine. For sake of this example, we'll use this.

4. Click the Add… button on the right side of this window. A second window will open, prompting you to select a driver for which you want to set up a data source. Choose the Microsoft Access Driver (*.mdb) and click Finish.

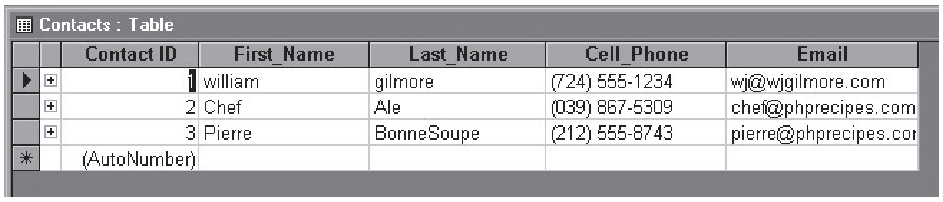
5. A new window will open, entitled "ODBC Microsoft Access Setup." You'll see a form text box labeled "Data Source Name." Enter a name relative to the Access database that you created. If you would like, enter a description in the text box directly below that of the Data Source Name.

6. Now click the Select… button displayed on the middle left of the window. An Explorer-style window will open, prompting you to search for the database that you would like to make accessible via ODBC.

7. Browse through the Windows directories to your database. When you find it, double-click it to select it. You will return to the ODBC Microsoft Access Setup window. You'll see the path leading to the selected database directly above the Select… button. Click OK.

8. That's it! Your Access database is now ODBC enabled.

Once you have completed this process, all you need to do is create the PHP script that will communicate via ODBC with the database. The script uses the Unified ODBC Functions introduced earlier to display all information in the Contacts table in the Access Contact Management database that I created using the Access wizard. However, before reviewing the script, take a look at the Contacts table in Figure 11-3, as seen in Access.



Contacts : Table

		Contact ID	First_Name	Last_Name	Cell_Phone	Email
▶	⊞	1	william	gilmore	(724) 555-1234	wj@wjgilmore.com
	⊞	2	Chef	Ale	(039) 867-5309	chef@phprecipes.com
	⊞	3	Pierre	BonneSoupe	(212) 555-8743	pierre@phprecipes.coi
✳		(AutoNumber)				

Figure 11-3. Contacts table as seen in MS Access

Now that you have an idea of what information will be extracted, take a look at the script. If you are unfamiliar with any of the functions, please refer to the introductory material at the beginning of this section. The outcome of Listing 11-7 is illustrated in Figure 11-4.

Listing 11-7: Using PHP's ODBC functions to interface with MS Access

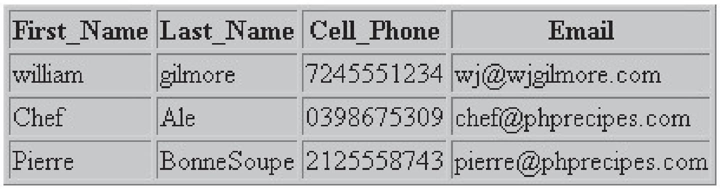
```
<?
// Connect to the ODBC datasource 'ContactDB'
$connect = odbc_connect("ContactDB","","")
                 or die("Couldn't connect to datasource.");

// form query statement
$query = "SELECT First_Name, Last_Name, Cell_Phone, Email FROM Contacts";

// prepare query statement
$result = odbc_prepare($connect,$query);

// execute query statement and display results
odbc_execute($result);
odbc_result_all($result,"border=1");
// We're done with the query results, so free memory
odbc_free_result($result);

// close connection
odbc_close($connect);
?>
```



First_Name	Last_Name	Cell_Phone	Email
william	gilmore	7245551234	wj@wjgilmore.com
Chef	Ale	0398675309	chef@phprecipes.com
Pierre	BonneSoupe	2125558743	pierre@phprecipes.com

Figure 11-4. Contacts table extracted and displayed to the Web browser

Pretty easy, right? The really great part is that this script is completely reusable with other ODBC-enabled database servers. As an exercise, go ahead and run through the above process, this time using a different database server. Rerun the script, and you will witness the same results as those shown in Figure 11-4.

Project: Create a Bookmark Repository

Probably the easiest way to build content on your site is to provide users with the capability of doing it for you. An HTML form is perhaps the most convenient way for accepting this information. Of course, the user-supplied information must also be processed and stored. In the project covered in the last chapter, you saw that this was easily accomplished with PHP and a text file. But what if you needed a somewhat more robust solution for storing the data? Sure, a text file works when storing relatively small and simple pieces of data, but chances are you will need a database to implement any truly powerful Web application. In this project, I explain how a MySQL database can be used to store information regarding Web sites. These sites are separated into categories to allow for more efficient navigation. Users can use an HTML form to enter information regarding their favorite sites, choosing a fitting category from those predefined by the administrator. Furthermore, users can go to an index page and click one of these predefined categories to view all of the sites under it.

The first thing you need to do is decide what site information should be stored in the MySQL database. To keep the project simple, I'll limit this to the following: site name, URL, category, date added, and description. Therefore, the MySQL table would look like the following:

```
mysql>create table bookmarks (
          category INT,
          site_name char(35),
          url char(50),
          date_added date,
          description char(254) );
```

There are a couple of points to be made regarding the bookmarks table. First of all, you may be curious as to why I chose to store the category information as an integer. After all, you want to use category names that are easily intelligible by the user, right? Don't worry, as you will soon create an array in an initialization file that will be used to create integer-to-category name mappings. This is useful because you may wish to modify or even delete a category in the future. Doing so is considerably easier if you use an array mapping to store the categories. Furthermore, an integer column will take up less information than would repetitive use of category names. Another point regarding the table is the choice to designate only 254 characters to the description. Depending on how extensive you would like the descriptions to be, you may want to change this column type to a medium text or even text. Check out the MySQL documentation for further information regarding possible column types.

The next step in creating this application will be to create an initialization file. Other than holding various global variables, two functions are defined within, `add_bookmark()` and `view_bookmark()`. The function `add_bookmark()` takes as input the user-entered form information and adds it to the database. The function `view_bookmark()` takes as input a chosen category and extracts all information from the database having that category, in turn displaying it to the browser. This file is shown in Listing 11-8, with accompanying comments.

Listing 11-8: Bookmark repository initialization file (init.inc)

```
<?
// file: init.inc
// purpose: provides global variables for use in bookmark project

// Default page title
$title = "My Bookmark Repository";

// Background color
$bg_color = "white";

// Posting date
$post_date = date("Ymd");

// bookmark categories
$categories = array(
                    "computers",
                    "entertainment",
                    "dining",
                    "lifestyle",
                    "government",
                    "travel");
```

Gilmore_11 12/5/00 10:24 AM Page 291

```
// MySQL Server Information
$host = "localhost";
$user = "root";
$pswd = "";

// database name
$database = "book";

// bookmark table name
$bookmark_table = "bookmarks";

// Table cell color
$cell_color = "#c0c0c0";

// Connect to the MySQL Server
@mysql_pconnect($host, $user, $pswd) or die("Couldn't connect to MySQL server!");

// Select the database
@mysql_select_db($database) or die("Couldn't select $database database!");

// function: add_bookmark()
// purpose: Add new bookmark to bookmark table.

function add_bookmark ($category, $site_name, $url, $description) {
     GLOBAL $bookmark_table, $post_date;

     $query = "INSERT INTO $bookmark_table
                    VALUES(\"$category\", \"$site_name\", \"$url\",
                    \"$post_date\", \"$description\")";

     $result = @mysql_query($query)
                     or die("Couldn't insert bookmark information!");

} // add_bookmark

// function: view_bookmark()
// purpose: View bookmarks following under
// the category $category.

function view_bookmark ($category) {

     GLOBAL $bookmark_table, $cell_color, $categories;
```

```
$query = "SELECT site_name, url, DATE_FORMAT(date_added,'%m-%d-%Y') AS
               date_added, description
               FROM $bookmark_table WHERE category = $category
               ORDER BY date_added DESC";

$result = @mysql_query($query);

print "<div align=\"center\"><table cellpadding=\"2\" cellspacing=\"1\"
border = \"0\" width = \"600\">";

print "<tr><td bgcolor=\"$cell_color\"><b>Category:
$categories[$category]</b></td></tr>";

if (mysql_numrows($result) > 0) :

     while ($row = mysql_fetch_array($result)) :

          print "<tr><td>";
          print "<b>".$row["site_name"]."</b> | Posted:
          ".$row["date_added"]."<br>";
          print "</td></tr>";

          print "<tr><td>";
          print "<a href = \"http://".$row["url"]."\">
                http://".$row["url"]."</a><br>";
          print "</td></tr>";

          print "<tr><td valign=\"top\">";
          print $row["description"]."<br>";
          print "</td></tr>";

          print "<tr><td><hr></td></tr>";

     endwhile;

else :
     print "<tr><td>There are currently no bookmarks falling under
            this category. Why don't you
            <a href=\"add_bookmark.php\">add one</a>?</td></tr>";

endif;

print "</table><a href=\"index.php\">Return to index</a> |";
```

```
     print "<a href=\"add_bookmark.php\">Add a bookmark</a></div>";

} // view_bookmark

?>
```

The next file, add_bookmark.php, provides the interface for adding new bookmark information to the database. Additionally, it calls the function `add_bookmark()` to process the user information. The file is shown in Listing 11-9.

Listing 11-9: The add_bookmark.php program

```
<html>
<?
INCLUDE("init.inc");
?>
<head>
<title><?=$title;?></title>
</head>

<body bgcolor="#ffffff" text="#000000" link="#808040" vlink="#808040"
alink="#808040">

<?
if (! $seenform) :
?>

<form action="add_bookmark.php" method="post">
<input type="hidden" name="seenform" value="y">

Category:<br>
<select name="category">
<option value="">Choose a category:
<?
while (list($key, $value) = each($categories)) :

     print "<option value=\"$key\">$value";

endwhile;
?>
</select><br>

Site Name:<br>
<input type="text" name="site_name" size="15" maxlength="30" value=""><br>
```

```
URL: (do <i>not</i> include "http://"!)<br>
<input type="text" name="url" size="35" maxlength="50" value=""><br>

Description:<br>
<textarea name="description" rows="4" cols="30"></textarea><br>
<input type="submit" value="submit">
</form>
<?
else :

     add_bookmark($category, $site_name, $url, $description);

     print "<h4>Your bookmark has been added to the repository. <a
href=\"index.php\">Click here</a> to return to the index.</h4>";

endif;
?>
```

When the user first requests this file, the interface shown in Figure 11-5 will be displayed to the browser.

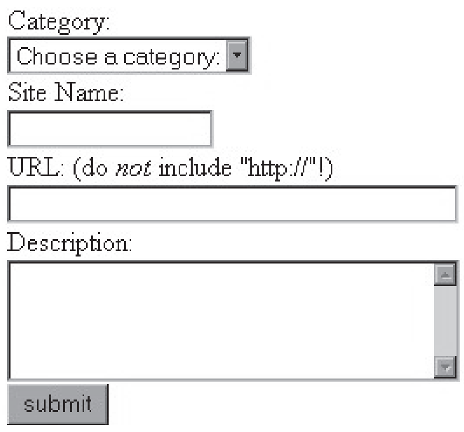


Figure 11-5. Form interface displayed in add_browser.php

Once a bookmark has been added to the database, the user is notified accordingly, and a link is provided for returning to the home page of the repository, entitled "index.php." This file is located in Listing 11-11, displayed below.

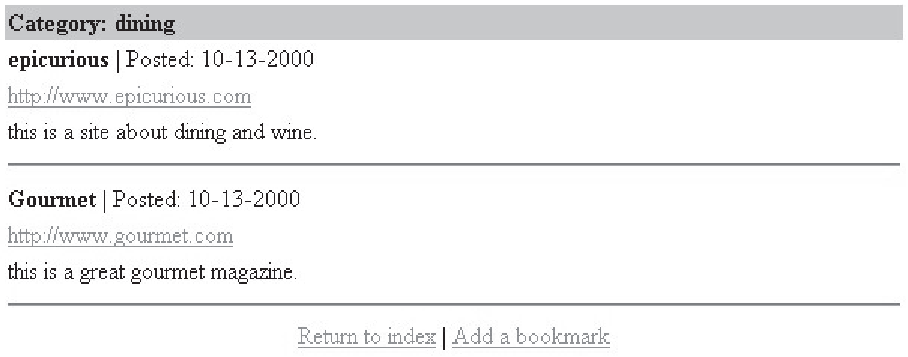
The next file, view_bookmark.php, simply calls the function `view_bookmark()`. The file is shown in Listing 11-10.

Listing 11-10: The view_bookmark.php program

```
<html>
<?
INCLUDE("init.inc");
?>
<head>
<title><?=$title;?></title>
</head>

<body bgcolor="<?=$bg_color;?>" text="#000000" link="#808040" vlink="#808040"
alink="#808040">
<?
view_bookmark($category);
?>
```

Assuming that you had entered a few sites under the dining category, executing view_bookmark.php would result in an interface similar to the one in Figure 11-6.



Category: dining

epicurious | Posted: 10-13-2000

http://www.epicurious.com

this is a site about dining and wine.

Gourmet | Posted: 10-13-2000

http://www.gourmet.com

this is a great gourmet magazine.

Return to index | Add a bookmark

Figure 11-6. Executing view_bookmark.php under the dining category

Finally, you need to provide an interface in which the user can choose the categorical list of bookmarks. I call this file index.php. The file is shown in Listing 11-11.

Listing 11-11: The index.php program

```
<html>
<?
INCLUDE("init.inc");
?>
<head>
<title><?=$title;?></title>
</head>
<body bgcolor="<?=$bg_color;?>" text="#000000" link="#808040" vlink="#808040"
alink="#808040">
<h4>Choose bookmark category to view:</h4>
<?
// cycle through each category and display appropriate link
while (list($key, $value) = each($categories)) :
     print "<a href = \"view_bookmark.php?category=$key\">$value</a><br>";
endwhile;
?>
<p>
<b><a href="add_bookmark.php">Add a new bookmark</a></b>
</body>
</html>
```

Assuming you don't change the default values in the `$categories` array in init.inc, the HTML in Listing 11-12 would be output to the browser when index.php is executed.

Listing 11-12: Output generated from index.php execution

```
<html>
<head>
<title></title>
</head>
<body bgcolor="white" text="#000000" link="#808040" vlink="#808040"
alink="#808040">
<h4>Choose bookmark category to view:</h4>
<a href = "view_bookmark.php?category=0">computers</a><br>
<a href = "view_bookmark.php?category=1">entertainment</a><br>
<a href = "view_bookmark.php?category=2">dining</a><br>
<a href = "view_bookmark.php?category=3">lifestyle</a><br>
<a href = "view_bookmark.php?category=4">government</a><br>
<a href = "view_bookmark.php?category=5">travel</a><br>
<p>
<b><a href="add_bookmark.php">Add a new bookmark</a></b>
</body>
</html>
```

Clicking any of the links in the preceding HTML source would result in the request of the file view_bookmark.php, which would then call the function `view_bookmark()`, passing the variable `$category` into it.

What's Next

This chapter covered a great deal of material, some of which may be the most important of the entire book for certain users. Database interfacing is certainly one of the most prominent features of the PHP language, as it can truly lend a hand to extending the functionality of a Web site. In particular, these topics were covered:

- Introduction to SQL
- An overview of PHP's predefined database support
- Case study of the MySQL database server
- PHP's predefined MySQL functions
- A simple search engine example
- A table resorting example
- Introduction to ODBC
- PHP's predefined ODBC functions
- Interfacing Microsoft Access with PHP
- A Web-based bulletin board

For those readers interested in developing large and truly dynamic Web sites with PHP, databasing will be an issue brought up time and time again. I recommend thoroughly reading not only the PHP documentation, but any other data warehousing resources available. As with most other technologies today, even the experts can't seem to learn enough about this rapidly advancing subject.

Next chapter, I delve into one of the more complicated topics of Web development: templates. In particular, I focus on how PHP can be used to create these templates, ultimately saving time and headaches when developing large-scale Web sites.