CHAPTER 31

Practical Database Queries

The previous chapter introduced PHP's PostgreSQL extension and demonstrated basic queries involving data selection. This chapter expands upon this foundational knowledge, demonstrating numerous concepts that you're bound to return to repeatedly while creating database-driven Web applications using the PHP language. In particular, you'll learn how to implement the following concepts:

- **A PostgreSQL database class**: Managing your database queries using a class not only results in cleaner application code, but also enables you to quickly and easily extend and modify query capabilities as necessary. This chapter presents a PostgreSQL database class implementation and provides several introductory examples so that you can familiarize yourself with its behavior.
- **Tabular output**: Listing query results in an easily readable format is one of the most commonplace tasks you'll implement when building database-driven applications. This chapter shows you how to create these listings by using HTML tables, and demonstrates how to link each result row to a corresponding detailed view.
- **Sorting tabular output**: Often, query results are ordered in a default fashion, by product name, for example. But what if the user would like to reorder the results using some other criteria, such as price? You'll learn how to provide table-sorting mechanisms that let the user search on any column.
- **Paged results**: Database tables often consist of hundreds, even thousands, of results. When large result sets are retrieved, it often makes sense to separate these results across several pages, and provide the user with a mechanism to navigate back and forth between these pages. This chapter shows you an easy way to do so.

The intent of this chapter isn't to imply that a single, solitary means exists for carrying out these tasks; rather, the intent is to provide you with some general insight regarding how you might go about implementing these features. If your mind is racing regarding how you can build upon these ideas after you've finished this chapter, the goal of this chapter has been met.

Sample Data

The examples found in this chapter use the `product` table created in the last chapter, reprinted here for your convenience:

```
CREATE TABLE product (
productid SERIAL,
productcode VARCHAR(8) NOT NULL UNIQUE,
name TEXT NOT NULL,
price NUMERIC(5,2) NOT NULL,
description TEXT NOT NULL,
PRIMARY KEY(productid)
);
```

Creating a PostgreSQL Database Class

Although we introduced PHP's PostgreSQL library in Chapter 30, you probably won't want to reference these functions directly within your scripts. Rather, you should encapsulate them within a class, and then use the class to interact with the database. Listing 31-1 offers a base functionality that one would expect to find in such a class.

Listing 31-1. *A PostgreSQL Data Layer Class (pgsql.class.php)*

```
<?php
class pgsql {
   private $linkid;      // PostgreSQL link identifier
   private $host;        // PostgreSQL server host
   private $user;        // PostgreSQL user
   private $passwd;      // PostgreSQL password
   private $db;          // PostgreSQL database
   private $result;      // Query result
   private $querycount;  // Total queries executed

   /* Class constructor. Initializes the $host, $user, $passwd
      and $db fields. */
   function __construct($host, $db, $user, $passwd) {
      $this->host = $host;
      $this->user = $user;
      $this->passwd = $passwd;
      $this->db = $db;
   }

   /* Connects to the PostgreSQL Database */
   function connect(){
      try{
         $this->linkid = @pg_connect("host=$this->host dbname=$this->db
            user=$this->user password=$this->passwd");
         if (! $this->linkid)
            throw new Exception("Could not connect to PostgreSQL server.");
      }
```

```
   catch (Exception $e) {
      die($e->getMessage());
   }
}

/* Execute database query. */
function query($query){
   try{
      $this->result = @pg_query($this->linkid,$query);
      if(! $this->result)
         throw new Exception("The database query failed.");
   }
   catch (Exception $e){
      echo $e->getMessage();
   }
   $this->querycount++;
   return $this->result;
}

/* Determine total rows affected by query. */
function affectedRows(){
   $count = @pg_affected_rows($this->linkid);
   return $count;
}

/* Determine total rows returned by query */
function numRows(){
   $count = @pg_num_rows($this->result);
   return $count;
}

/* Return query result row as an object. */
function fetchObject(){
   $row = @pg_fetch_object($this->result);
   return $row;
}

/* Return query result row as an indexed array. */
function fetchRow(){
   $row = @pg_fetch_row($this->result);
   return $row;
}

/* Return query result row as an associated array. */
function fetchArray(){
   $row = @pg_fetch_array($this->result);
   return $row;
}
```

```
   /* Return total number of queries executed during
      lifetime of this object. Not required, but
      interesting nonetheless. */
   function numQueries(){
      return $this->querycount;
   }
}
?>
```

The code found in Listing 31-1 should be easy to comprehend at this point in the book, which is why it is light on comments. However, you should note in particular one point that's pertinent to the output of exceptions. To keep matters simple, we've used a `die()` statement for outputting the exception that is specific to connecting to the database server; in contrast, failed queries will not be fatally returned. Depending on your particular needs, this implementation might not be exactly suitable, but it should work just fine for the purposes of this book. The remainder of this section is devoted to several examples, each aimed to better familiarize you with use of the PostgreSQL class.

Why Use the PostgreSQL Database Class?

If you're new to object-oriented programming, you may still be unconvinced that you should use the class-oriented approach, and may be thinking about directly embedding the PostgreSQL functions in the application code. In hopes of remedying such reservations, this section illustrates the advantages of the class-based strategy by providing two examples. Both examples implement the simple task of querying the `company` database for a particular product name and price. However, the first example (shown next) does so by calling the PostgreSQL functions directly from the application code, whereas the second example uses the PostgreSQL class library shown in Listing 31-1 in the prior section.

```
<?php
   /* Connect to the database server /*
   $linkid = @pg_pconnect("host=localhost dbname=company user=rob
      password=secret") or
      die("Could not connect to the PostgreSQL server.");
   /* Execute the query */
   $result = @pg_query("SELECT name,price FROM product ORDER BY
      productid") or die("The database query failed.");
   /* Output the results */
   while($row = pg_fetch_object($result))
      echo "$row->name (\$$row->price)<br />";
?>
```

The next example uses the PostgreSQL class:

```
<?php
   include "pgsql.class.php";
   /* Create a new pgsql object */
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   /* Connect to the database server and select a database */
```

```
   $pgsqldb->connect();
   /* Execute the query */
   $pgsqldb->query("SELECT name, price FROM product ORDER BY productid");
   /* Output the results */
   while ($row = $pgsqldb->fetchObject())
      echo "$row->name (\$$row->price)<br />";
?>
```

Both examples return output similar to the following:

```
PHP T-Shirt ($12.99)
PostgreSQL Coffee Cup ($4.99)
```

The following list summarizes the numerous advantages of using the object-oriented approach over calling the PostgreSQL functions directly from the application code:

- The code is cleaner. Intertwining logic, data queries, and error messages results in jumbled code.
- Because the database access code is encapsulated in the class, changing database types is a trivial task.
- Encapsulating the error messages in the class allows for easy modification and internationalization.
- Any changes to the PostgreSQL API can easily be implemented through the class. Imagine having to manually modify PostgreSQL function calls spread throughout 50 application scripts! This is not just a theoretical argument; PHP's PostgreSQL API did change as recently as the PHP 4.2 release, and developers who made extensive use of the direct PostgreSQL functions were forced to deal with this problem.

Executing a Simple Query

Before delving into somewhat more complex topics involving the PostgreSQL database class, a few introductory examples should be helpful. For starters, the following example shows you how to connect to the database server and retrieve some data using a simple query:

```
<?php
   include "pgsql.class.php";

   /* Create new pgsql object */
   $pgsqldb = new pgsql("localhost","company","rob","secret");

   /* Connect to the database server and select a database */
   $pgsqldb->connect();

   /* Query the database */
   $pgsqldb->query("SELECT name, price FROM product
                    WHERE productcode = 'tshirt01'");
```

```
   /* Retrieve the query result as an object */
   $row = $pgsqldb->fetchObject();

   /* Output the data */
   echo "$row->name (\$$row->price)";
?>
```

This example returns:

```
PHP T-Shirt ($12.99)
```

Retrieving Multiple Rows

Now consider a slightly more involved example. The following script retrieves all rows from the `product` table, ordering by name:

```
<?php
   include "pgsql.class.php";

   /* Create new pgsql object */
   $pgsqldb = new pgsql("localhost","company","rob","secret");

   /* Connect to the database server and select a database */
   $pgsqldb->connect();

   /* Query the database */
   $pgsqldb->query("SELECT name, price FROM product ORDER BY name");

   /* Output the data */
   while ($row = $pgsqldb->fetchObject())
      echo "$row->name (\$$row->price)<br />";
?>
```

This returns:

```
Linux Hat ($8.99)
PHP Coffee Cup ($3.99)
Ruby Hat ($16.99)
```

Counting Queries

This section illustrates how easy it is to extend the capabilities of a data class, just one of the advantages of embedding the PostgreSQL functionality in this fashion (other advantages are listed in the earlier section, "Why Use the PostgreSQL Database Class?"). The `numQueries()` method that appears at the end of Listing 31-1 retrieves the values of the private field `$querycount`. This field is incremented every time a query is executed, allowing you to keep track of the total number

of queries executed throughout the lifetime of an object. The following example executes two queries and then outputs the number of queries executed:

```
<?php
   include "pgsql.class.php";

   // Create a new pgsql object
   $pgsqldb = new pgsql("localhost","company","rob","secret");

   // Connect to the database server and select a database
   $pgsqldb->connect();

   // Execute a few queries
   $query = "SELECT name, price FROM product ORDER BY name";
   $pgsqldb->query($query);
   $query2 = "SELECT name, price FROM product
              WHERE productcode='tshirt01'";
   $pgsqldb->query($query2);

   // Output the total number of queries executed.
   echo "Total number of queries executed: ".$pgsqldb->numQueries();

?>
```

The example returns the following:

```
Total number of queries executed: 2
```

Tabular Output

Viewing retrieved database data in a coherent, user-friendly fashion is key to the success of a Web application. HTML tables have been used for years to satisfy this need for uniformity, for better or for worse. Because this functionality is so commonplace, it makes sense to encapsulate this functionality in a function, and call that function whenever database results should be formatted in this fashion. This section demonstrates one way to accomplish this.

For reasons of convenience, we'll create this function in the format of a method and add it to the PostgreSQL data class. To facilitate this method, we also need to add two more helper methods, one for determining the number of fields in a result set, and another for determining each field name:

```
/* Return the number of fields in a result set */
function numberFields() {
   $count = @pg_num_fields($this->result);
   return $count;
}
```

```
/* Return a field name given an integer offset. */
function fieldName($offset){
   $field = @pg_field_name($this->result, $offset);
   return $field;
}
```

We'll use these methods along with other methods in the PostgreSQL data class to create an easy and convenient method named `getResultAsTable()`, used to output table-encapsulated results. This method is highly useful for two reasons in particular. First, it automatically converts the field names into the table headers. Second, it automatically adjusts to a number of fields found in the query. It's a one size fits all solution for formatting of this sort. The method is presented in Listing 31-2.

Listing 31-2. *The getResultAsTable() Method*

```
function getResultAsTable() {
   if ($this->numrows() > 0) {
      // Start the table
      $resultHTML = "<table border='1'>\n<tr>\n";

      // Output the table headers
      $fieldCount = $this->numberFields();
      for ($i=0; $i < $fieldCount; $i++){
         $rowName = $this->fieldName($i);
         $resultHTML .= "<th>$rowName</th>";
      } // end for

      // Close the row
      $resultHTML .= "</tr>\n";

      // Output the table data
      while ($row = $this->fetchRow()){
         $resultHTML .= "<tr>\n";
         for ($i = 0; $i < $fieldCount; $i++)
            $resultHTML .= "<td>".htmlentities($row[$i])."</td>";
         $resultHTML .= "</tr>\n";
      }

      // Close the table
      $resultHTML .= "</table>";
   }
   else {
      $resultHTML = "<p>No Results Found</p>";
   }
   return $resultHTML;
}
```

Using `getResultAsTable()` is easy, as demonstrated in the following code snippet:

```
<?php
   include "pgsql.class.php";

   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Execute the query
   $pgsqldb->query('SELECT name as "Product",
                    price as "Price",
                    description as "Description" FROM product");

   // Return the result as a table
   echo $pgsqldb->getResultAsTable();
?>
```

Example output is displayed in Figure 31-1.

Product	Price	Description
Linux Hat	8.99	A hat that says 'I Love Linux'.
Ruby Coffee Cup	5.99	Sparkle like a ruby in the morning with this Ruby coffee cup
PostgreSQL Coffee Cup	4.99	Display your database allegiance to the morning crew with this 11 oz. coffee cup.
Ruby Hat	16.99	Spread the good vibe by wearing this Ruby baseball cap with pride. The text states, "Ruby is for lovers"
PHP Coffee Cup	3.99	Relax, you have a PHP powered breakfast with this 11 oz. coffee cup
PHP T-Shirt	12.99	Weat this PHP T-shirt with pride

Figure 31-1. *Creating table-formatted results*

Linking to a Detailed View

Often a user will want to do more with the results than just view them. For example, the user might want to learn more about a particular product found in the result, or he might want to add a product to his shopping cart. An interface that offers such capabilities is presented in Figure 31-2.

Product	Price	Actions
Linux Hat	8.99	View Detailed | Add To Cart
PHP Coffee Cup	3.99	View Detailed | Add To Cart
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart
Ruby Coffee Cup	5.99	View Detailed | Add To Cart
Ruby Hat	16.99	View Detailed | Add To Cart

Figure 31-2. *Offering actionable options in the table output*

As it currently stands, the `getResultsAsTable()` method doesn't offer the ability to accompany each row with actionable options. This section shows you how to modify the code to provide this functionality. Before diving into the code, however, a few preliminary points are in order.

For starters, we want to be able to pass in actions of varying purpose and number. That said, we'll pass the actions in as a string to the function. However, because the actions will be included at the end of each line, we won't know what primary key to tack on to each action until the row has been rendered. This is essential, because the destination script needs to know which item is targeted. Therefore, run-time replacement on a predefined string must occur when the rows are being formatted for output. We'll use the string `VALUE` for this purpose. An example action string follows:

```
$actions = '<a href="viewdetail.php?productid=VALUE">View Detailed</a> |
            <a href="addtocart.php?productid=VALUE">Add to Cart</a>';
```

Of course, this also implies that an identifying key must be included in the query. The `product` table's primary key is `productid`. In addition, this key should be placed first in the query, because the script needs to know which field value to serve as a replacement for `VALUE`. Finally, because you probably don't want the `productid` to appear in the formatted table, the counters used to output the table header and data need to be incremented by 1.

The updated `getResultAsTable()` method follows. For your convenience, those lines that have either changed or are new appear in bold.

```
function getResultAsTable($actions){

   if ($this->numrows() > 0) {
      // Start the table
      $resultHTML = "<table border='1'>\n<tr>\n";

      // Output the table header
      $fieldCount = $this->numberFields();
      for ($i=1; $i < $fieldCount; $i++) {
         $rowName = $this->fieldName($i);
         $resultHTML .= "<th>$rowName</th>\n";
      } // end for

      $resultHTML .= "<th>Actions</th></tr>\n";

      // Output the table data
      while ($row = $this->fetchRow()) {
         $resultHTML .= "<tr>\n";
         for ($i=1; $i < $fieldCount; $i++)
            $resultHTML .= "<td>".htmlentities($row[$i])."</td>\n";

      // Replace VALUE with the correct primary key
      $action = str_replace("VALUE", $row[0], $actions);
      // Add the action cell to the end of the row
      $resultHTML .= "<td nowrap> $action</td>\n</tr>\n";

      } // end while
```

```
      // Close the table
      $resultHTML .= "</table>\n";
   } else {
      $resultHTML = "No results found";
   }
   return $resultHTML;
}
```

The process for executing this method is almost identical to the original. The key difference is that this time you have the option of including actions:

```
<?php
   include "pgsql.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Query the database
   $pgsqldb->query('SELECT productid, name as "Product",
                    price as "Price" FROM product ORDER BY name');

   $actions='<a href="viewdetail.php?productid=VALUE">View
             Detailed</a> |
             <a  href="addtocart.php?productid=VALUE">Add To Cart</a>';

   echo $pgsqldb->getResultAsTable($actions);

?>
```

Executing this code will produce output similar to that found in Figure 31-2.

Sorting Output

When displaying output, it makes sense to order the information using criteria that are most helpful to the user. For example, if the user wants to view a list of all products in the `product` table, ordering the products in ascending alphabetical order makes sense. However, some users may want to order the information by other criteria, price for example. Often, such mechanisms are implemented by linking listing headers, such as the table headers used in the previous examples. Clicking any of these links will cause the table data to be sorted using that header as criterion. This section demonstrates this concept, again modifying the most recent version of the `getResultsAsTable()` method. However, only one line requires modification, specifically:

```
$resultHTML .= "<th>$rowName</th>";
```

This line is changed to the following:

```
$sqlPosition = $i+1;
$resultHTML .= "<th>
                <a href=\"".$_SERVER['PHP_SELF']."?sort=$rowName\">$rowName</a>
                </th>";
```

The executing code looks quite similar to that used in previous examples, except now a dynamic SORT clause must be inserted in the query. A ternary operator, introduced in Chapter 3, is used to determine whether the user has clicked one of the header links:

```
$sort = (isset($_GET['sort'])) ? $_GET['sort'] : "name";
```

If a sort parameter has been passed via the URL, that value will be the sorting criteria. Otherwise, a default of name is used.

```
$pgsqldb->query("SELECT productid, name as Product,
                 price as Price FROM product ORDER BY $sort ASC");
```

The complete executing code follows:

```
<?php
   include "pgsql.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Determine what kind of sort request has been submitted
   // By default this is set to sort name
   $sort = (isset($_GET['sort'])) ? $_GET['sort'] : 'name';

   // Query the database
   $pgsqldb->query("SELECT productid, name as \"Product\",
                    price as \"Price\" FROM product ORDER BY $sort ASC");

   $actions = '<a href="viewdetail.php?productid=VALUE">View Detailed</a> | <a
href="addtocart.php?productid=VALUE">Add
To Cart</a>';

   echo $pgsqldb->getResultAsTable($actions);
?>
```

■ **Note** This example strives for simplicity over security. If this were a real application, you would not want to directly assign the $sort variable to your SQL statement, but instead would do some type of integrity checking on the value passed in, to make sure your users have not tried to send across malicious data. Keep this in mind when viewing the following examples as well as when you are writing your own code.

Loading the script for the first time results in the output being sorted by name. Example output is shown in Figure 31-3.

Product	Price	Actions
Linux Hat	8.99	View Detailed | Add To Cart
PHP Coffee Cup	3.99	View Detailed | Add To Cart
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart
Ruby Coffee Cup	5.99	View Detailed | Add To Cart
Ruby Hat	16.99	View Detailed | Add To Cart

Figure 31-3. *The product table output sorted by the default name*

Clicking the `Price` header re-sorts the output. This sorted output is shown in Figure 31-4.

Product	Price	Actions
PHP Coffee Cup	3.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart
Ruby Coffee Cup	5.99	View Detailed | Add To Cart
Linux Hat	8.99	View Detailed | Add To Cart
PHP T-Shirt	12.99	View Detailed | Add To Cart
Ruby Hat	16.99	View Detailed | Add To Cart

Figure 31-4. *The product table output sorted by price*

Creating Paged Output

If you've perused any e-commerce sites or search engines, you're familiar with the concept of separating output into several pages. This feature is convenient not only to enhance readability, but also to further optimize page loading. You might be surprised to learn that adding this feature to your Web site is a trivial affair. This section demonstrates how this is accomplished.

This feature depends in part on two SQL clauses: `LIMIT` and `OFFSET`. The `LIMIT` clause is used to specify the number of rows returned, and the `OFFSET` clause specifies a starting point to begin counting from. In general, the format looks like this:

```
LIMIT number_rows OFFSET starting_point
```

For example, if you want to limit returned query results to just the first five rows, you could construct the following query:

```
SELECT name, price FROM product ORDER BY name ASC LIMIT 5;
```

If you intend to start from the very first row, this is the same as:

```
SELECT name, price FROM product ORDER BY name ASC LIMIT 5 OFFSET 0;
```

However, to start from the sixth row of the result set, you would construct the following query:

```
SELECT name, price FROM product ORDER BY name ASC LIMIT 5 OFFSET 5;
```

Because this syntax is so convenient, you need to determine only three variables to create a mechanism for paging throughout the results:

- **Number of entries per page**: This is entirely up to you. Alternatively, you could easily offer the user the ability to customize this variable. This value is passed into the `number_rows` component of the `LIMIT` clause.
- **Row offset**: This value depends on what page is presently loaded. This value is passed by way of the URL so that it can be passed to the `OFFSET` clause. You'll see how to calculate this value in the following code.
- **Total number of rows in the result set**: This is required knowledge because the value is used to determine whether the page needs to contain a next link.

Interestingly, no modification to the PostgreSQL database class is required. Because this concept seems to cause quite a bit of confusion, we'll review the code first, and then see the example in its entirety in Listing 31-3. The first section is typical of any script using the PostgreSQL data class:

```
<?php
   include "pgsqldb.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();
```

The maximum number of entries that should appear on each paged result is then specified:

```
$pagesize = 2;
```

Next, a ternary operator determines whether the `$_GET['recordstart']` parameter has been passed by way of the URL. This parameter determines the offset from which the result set should begin. If this parameter is present, it's assigned to `$recordstart`; otherwise, `$recordstart` is set to `0`:

```
$recordstart = (isset($_GET['recordstart'])) ? $_GET['recordstart'] : 0;
```

Next, the database query is executed and the data is displayed. Note that the record offset is set to `$recordstart`, and the number of entries to retrieve is set to `$pagesize`:

```
$pgsqldb->query("SELECT name, price FROM product
                 ORDER BY name LIMIT $pagesize OFFSET $recordset");
// Output the result set
$actions = '<a href="viewdetail.php?productid=VALUE">View Detailed</a> |
<a href="addtocart.php?productid=VALUE">Add To Cart</a>';

echo $pgsqldb->getResultAsTable($actions);
```

Next, we need to determine the total number of rows available, accomplished by removing the `LIMIT` and `OFFSET` clauses from the original query. However, to optimize the query, we use the `count(*)` function rather than retrieve the complete result set:

```
$pgsqldb->query("SELECT count(*) FROM product");
$row = $pgsqldb->fetchObject();
$totalrows = $row->count;
```

Finally, the previous and next links are created. The previous link is created only if the record offset, `$recordstart`, is greater than `0`. The next link is created only if some records remain to be retrieved, meaning `$recordstart` + `$pagesize` must be less than `$totalrows`.

```
// Create the 'previous' link
if ($recordstart > 0) {
   $prev = $recordstart - $pagesize;
   $url = $_SERVER['PHP_SELF']."?recordstart=$prev";
   echo "<a href=\"$url\">Previous Page</a> ";
}

// Create the 'next' link
if ($totalrows > ($recordstart + $pagesize) {
   $next = $recordstart + $pagesize;
   $url = $_SERVER['PHP_SELF']."?recordstart=$next";
   echo "<a href=\"$url\">Next Page</a>";
}
```

Sample output is shown in Figure 31-5. The complete code listing is presented in Listing 31-3.

Product	Price	Actions
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart

Previous Page Next Page

Figure 31-5. *Creating paged results (two results per page)*

Listing 31-3. *Paging Database Results*

```
<?php
   include "pgsql.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Set the number of entries per page
   $pagesize = 2;

   // What is our record offset?
   $recordstart = (isset($_GET['recordstart'])) ? $_GET['recordstart'] : 0;

   // Execute the SELECT query, including the LIMIT and OFFSET clauses
   $pgsqldb->query("SELECT productid, name as \"Product\",
                    price as \"Price\" FROM product
                    ORDER BY name LIMIT $pagesize OFFSET $recordstart");
```

```
   // Output the result set
   $actions = '<a href="viewdetail.php?productid=VALUE">View
   Detailed</a> | <a href="addtocart.php?productid=VALUE">Add To
   Cart</a>';

   echo $pgsqldb->getResultAsTable($actions);

   // Determine whether additional rows are available
   $pgsqldb->query("SELECT count(*) FROM product");
   $row = $pgsqldb->fetchObject();
   $totalrows = $row->count;

   // Create the 'previous' link
   if ($recordstart > 0) {
      $prev = $recordstart - $pagesize;
      $url = $_SERVER['PHP_SELF']."?recordstart=$prev";
      echo "<a href=\"$url\">Previous Page</a> ";
   }

   // Create the 'next' link
   if ($totalrows > ($recordstart + $pagesize) {
      $next = $recordstart + $pagesize;
      $url = $_SERVER['PHP_SELF']."?recordstart=$next";
      echo "<a href=\"$url\">Next Page</a>";
   }
?>
```

Listing Page Numbers

If you have several pages of results, the user might wish to traverse them in nonlinear order. For example, the user might choose to jump from page one to page three, then page six, then back to page one again. Thankfully, providing users with a linked list of page numbers is surprisingly easy. Building on Listing 31-3, you start by determining the total number of pages, and assigning that value to `$totalpages`. You determine the total number of pages by dividing the total result rows by the chosen page size, and round upwards using the `ceil()` function:

```
$totalpages = ceil($totalrows / $pagesize);
```

Next, you determine the current page number, and assign it to `$currentpage`. You determine the current page number by dividing the present record offset (`$recordstart`) by the chosen page size (`$pagesize`), and adding one to account for the fact that `LIMIT` offsets start with `0`:

```
$currentpage = ($recordstart / $pagesize ) +1;
```

Next, create a function titled `pageLinks()` and pass to it the following four parameters:

- `$totalpages`: The total number of pages, stored in the `$totalpages` variable.
- `$currentpage`: The current page, stored in the `$currentpage` variable.

- `$pagesize`: The chosen page size, stored in the `$pagesize` variable.
- `$parameter`: The name of the parameter used to pass the record offset by way of the URL. Thus far, `$recordstart` has been used, so we'll stick with that in the following example.

The `pageLinks()` function follows:

```
function pageLinks($totalpages, $currentpage, $pagesize, $parameter) {
   // Start at page one
   $page = 1;

   // Start at record 0
   $recordstart = 0;

   // Initialize page links
   $pageLinks = '';

   while ($page <= $totalpages) {
      // Link the page if it isn't the current one
      if ($page != $currentpage) {
         $pageLinks .= "<a href=\"".$_SERVER['PHP_SELF'].
         "?$parameter=$recordstart\">$page</a> ";
      // If the current page, just list the number
      }
      else {
         $pageLinks .= "$page ";
      }

      // Move to the next record delimiter
      $recordstart += $pagesize;
      $page++;
   }
   return $pageLinks;
}
```

Finally, you call the function like so:

```
echo "<p>Pages: ".
      pageLinks($totalpages, $currentpage, $pagesize, "recordstart").
      "</p>";
```

Sample output of the page listing, combined with other components introduced throughout this chapter, is shown in Figure 31-6.

Product	Price	Actions
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart

Previous Page Next Page

Pages: 1 2 3

Figure 31-6. *Generating a numbered list of page results*

Summary

This chapter offered insight into some of the most common general tasks you'll encounter when developing data-driven applications. The chapter started by providing a PostgreSQL data class and offering some basic usage examples involving this class. Next, you learned a convenient and easy method for outputting data results in a tabular format, and then learned how to add actionable options for each output data row. Building upon this material, you saw how to sort output based on a given table field. Finally, you learned how to spread query results across several pages and create linked page listings, enabling the user to navigate the results in a nonlinear fashion.

The next chapter introduces PostgreSQL's implementation of views and rules, which help you to implement and maintain your full data model.