In a multiple-tier PHP software architecture, only one layer (comprising one set of PHP programs) accesses the persistence layer, which is otherwise known as the database or filesystem. That's the accessor layer.

When, as is usually the case, the persistence layer is a database server, the accessor layer has a lot to do with commands that manipulate the database. In other words, the accessor programs contain lots of structured query language (SQL) statements that add data to the database, remove data from the database, modify data already in the database, and extract particular sets of data from the database.

However, because the accessor layer must deal with the peculiarities of the database server to which it has to connect, it's also concerned with connection protocols. Fortunately, the hard work of establishing a database connection has already been done for you by the PEAR DB in most cases (and, truth be told, it isn't really strenuous even without PEAR). By using PEAR DB, you simply define a datasource name (DSN) in the PHP programs that make up your application's accessor layer, and let PEAR DB make the connection. If you need to change database servers, the job is relatively easy and not too prone to error, because there's little code to change.

This chapter explains the ins and outs of SQL as it pertains to typical software applications. It also shows how to write PHP programs that use the PEAR DB library to access a database, the NuSOAP (simple object access protocol) library to expose functions as Web services, and SQL to manipulate the database that was created in Chapter 6.

7.1 Extracting Data from the Database

A database is useless unless it's possible to extract data from it in a meaningful way. Specifically, we need to be able to tell the database server what data we want it to extract
from the database, and how we want it presented. In the simplest case, we might want to see all columns in all rows in a certain table. In slightly more complicated situations, we might want to extract only certain rows, or only certain columns, or only certain columns of certain rows. You get the idea. This section explains how to use SQL SELECT queries effectively.

### 7.1.1 Using SELECT Queries

You use a SELECT query to get information out of a database. SELECT queries do not make any changes to the tables upon which they operate. To use a SELECT query, you simply specify which records you want to extract, and from which tables.

**Extracting All Columns from All Rows**

The simplest example of this is the ubiquitous "show all records" query:

```
SELECT * FROM demo;
```

That statement returns all columns and rows from the table named demo:

<table>
<thead>
<tr>
<th>id</th>
<th>lastName</th>
<th>firstName</th>
<th>yearOfBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Baldoni</td>
<td>Burt</td>
<td>1956</td>
</tr>
<tr>
<td>2</td>
<td>Carey</td>
<td>Casey</td>
<td>1982</td>
</tr>
<tr>
<td>3</td>
<td>Delaney</td>
<td>Dave</td>
<td>1977</td>
</tr>
<tr>
<td>4</td>
<td>Estacio</td>
<td>Elena</td>
<td>1934</td>
</tr>
<tr>
<td>5</td>
<td>Farquhar</td>
<td>Fred</td>
<td>1961</td>
</tr>
<tr>
<td>6</td>
<td>Ghiotto</td>
<td>Gail</td>
<td>1966</td>
</tr>
<tr>
<td>7</td>
<td>Hertenberger</td>
<td>Hal</td>
<td>1954</td>
</tr>
<tr>
<td>8</td>
<td>Imrie</td>
<td>Iggy</td>
<td>1957</td>
</tr>
<tr>
<td>9</td>
<td>Jindi</td>
<td>Jerry</td>
<td>1957</td>
</tr>
<tr>
<td>10</td>
<td>Karakis</td>
<td>Kelly</td>
<td>1988</td>
</tr>
<tr>
<td>11</td>
<td>Loudoun</td>
<td>Laurie</td>
<td>1971</td>
</tr>
<tr>
<td>12</td>
<td>McClellan</td>
<td>Midge</td>
<td>1965</td>
</tr>
<tr>
<td>13</td>
<td>MacClellan</td>
<td>Mike</td>
<td>1945</td>
</tr>
<tr>
<td>14</td>
<td>MacClellan</td>
<td>Mabel</td>
<td>1945</td>
</tr>
<tr>
<td>15</td>
<td>McClellan</td>
<td>Marcel</td>
<td>1958</td>
</tr>
<tr>
<td>16</td>
<td>Nacelle</td>
<td>Nancy</td>
<td>1962</td>
</tr>
<tr>
<td>17</td>
<td>Odd</td>
<td>Otis</td>
<td>1962</td>
</tr>
</tbody>
</table>

In that SQL statement, the star (*) stands for "all columns," and FROM is an SQL keyword that identifies the table against which the query is to run.
Extracting Only Certain Columns from All Rows

A query like the preceding one can easily return an unwieldy quantity of rows. A slightly more complicated syntax enables us to specify that we want to see only certain columns from the demo database:

```
SELECT firstName, lastName FROM demo;
```

In that statement, we limit the data returned in terms of columns. Even if there are many columns in demo in addition to firstName and lastName, only those two columns are returned for each row, like this:

<table>
<thead>
<tr>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burt</td>
<td>Baldoni</td>
</tr>
<tr>
<td>Casey</td>
<td>Carey</td>
</tr>
<tr>
<td>Dave</td>
<td>Delaney</td>
</tr>
<tr>
<td>Elena</td>
<td>Estacio</td>
</tr>
<tr>
<td>Fred</td>
<td>Farquhar</td>
</tr>
<tr>
<td>Gail</td>
<td>Ghiotto</td>
</tr>
<tr>
<td>Hal</td>
<td>Hertenberger</td>
</tr>
<tr>
<td>Iggy</td>
<td>Imrie</td>
</tr>
<tr>
<td>Jerry</td>
<td>Jindi</td>
</tr>
<tr>
<td>Kelly</td>
<td>Karakis</td>
</tr>
<tr>
<td>Laurie</td>
<td>Loudoun</td>
</tr>
<tr>
<td>Midge</td>
<td>McClellan</td>
</tr>
<tr>
<td>Mike</td>
<td>MacClellan</td>
</tr>
<tr>
<td>Mabel</td>
<td>MacClellan</td>
</tr>
<tr>
<td>Marcel</td>
<td>McClellan</td>
</tr>
<tr>
<td>Nancy</td>
<td>Nacelle</td>
</tr>
<tr>
<td>Otis</td>
<td>Odd</td>
</tr>
</tbody>
</table>

That's an important fact to note: The preceding statement and its predecessor return an equal number of rows. They differ only in the number of columns returned. Note also that the second statement, in which column names were specified, displayed the columns in the sequence specified in the query—the opposite of the default sequence.

Extracting Only Rows with Unique Values in Specified Columns

Note that SELECT will happily return duplicate values. This may or may not be what you want. If you require only a list of the distinct values that exist in a particular column, you can do something like this:

```
SELECT DISTINCT lastName FROM demo;
```
That line returns a list of unique lastName values that are represented in the database, like this:

<table>
<thead>
<tr>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baldoni</td>
</tr>
<tr>
<td>Carey</td>
</tr>
<tr>
<td>Delaney</td>
</tr>
<tr>
<td>Estacio</td>
</tr>
<tr>
<td>Farquhar</td>
</tr>
<tr>
<td>Ghiotto</td>
</tr>
<tr>
<td>Hertenberger</td>
</tr>
<tr>
<td>Imrie</td>
</tr>
<tr>
<td>Jindi</td>
</tr>
<tr>
<td>Karakis</td>
</tr>
<tr>
<td>Loudoun</td>
</tr>
<tr>
<td>MacClellan</td>
</tr>
<tr>
<td>McClellan</td>
</tr>
<tr>
<td>Nacelle</td>
</tr>
<tr>
<td>Odd</td>
</tr>
</tbody>
</table>

Note that McClellan and MacClellan, which appear in the table several times, appears only once in the result set of the statement above.

**Extracting Only Certain Rows**

In an earlier example, you got an idea of how to return all rows in a table, while limiting the results to certain columns. It's also possible to do the converse: Return all columns, but only certain rows. This is the reason behind adding a WHERE clause to the end of a SELECT clause.

The simplest application of the WHERE clause is in finding strict matches, as with this example:

```
SELECT * FROM demo WHERE lastName='Ghiotto';
```

This returns the following:

<table>
<thead>
<tr>
<th>lastName</th>
<th>firstName</th>
<th>yearOfBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghiotto</td>
<td>Gail</td>
<td>1966</td>
</tr>
</tbody>
</table>

Note that the statement causes the return of all columns of rows in which the lastName columns matches "Ghiotto" exactly. In this case, it happens to be just one. Such a match is strictly case sensitive.
### 7.1 Extracting Data from the Database

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Greater than or equal to one number, and less than or equal to another</td>
</tr>
<tr>
<td>LIKE</td>
<td>Matches a pattern that can include wildcards</td>
</tr>
</tbody>
</table>

**Table 7.1: SQL Comparison Operators.**

You can describe other kinds of matches with a series of operators that work with the **WHERE** keyword. These are shown in Table 7.1.

Most of the operators work best (or, at least, most intuitively) with numeric column values. This statement:

```
SELECT firstName, lastName, yearOfBirth FROM demo WHERE yearOfBirth <= 1975;
```

returns specified columns from all rows in which the value in the `yearOfBirth` column is less than or equal to 1974, effectively generating a list of people born in or before that year:

<table>
<thead>
<tr>
<th>firstName</th>
<th>lastName</th>
<th>yearOfBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burt</td>
<td>Baldoni</td>
<td>1956</td>
</tr>
<tr>
<td>Elena</td>
<td>Estacio</td>
<td>1934</td>
</tr>
<tr>
<td>Fred</td>
<td>Farquhar</td>
<td>1961</td>
</tr>
<tr>
<td>Gail</td>
<td>Ghiotto</td>
<td>1966</td>
</tr>
<tr>
<td>Hal</td>
<td>Hertenberger</td>
<td>1954</td>
</tr>
<tr>
<td>Iggy</td>
<td>Imrie</td>
<td>1957</td>
</tr>
<tr>
<td>Jerry</td>
<td>Jindi</td>
<td>1957</td>
</tr>
<tr>
<td>Laurie</td>
<td>Loudoun</td>
<td>1971</td>
</tr>
<tr>
<td>Midge</td>
<td>McClellan</td>
<td>1965</td>
</tr>
<tr>
<td>Mike</td>
<td>MacClellan</td>
<td>1945</td>
</tr>
<tr>
<td>Mabel</td>
<td>MacClellan</td>
<td>1945</td>
</tr>
<tr>
<td>Marcel</td>
<td>McClellan</td>
<td>1958</td>
</tr>
<tr>
<td>Nancy</td>
<td>Nacelle</td>
<td>1962</td>
</tr>
<tr>
<td>Otis</td>
<td>Odd</td>
<td>1962</td>
</tr>
</tbody>
</table>

The LIKE operator is handiest when you are uncertain about spellings. This statement:

```
SELECT firstName, lastName, yearOfBirth FROM demo WHERE lastName LIKE 'McClellan';
```
returns rows in which the lastName column value is either “McClellan” or “MacClellan”

<table>
<thead>
<tr>
<th>firstName</th>
<th>lastName</th>
<th>yearOfBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Midge</td>
<td>McClellan</td>
<td>1965</td>
</tr>
<tr>
<td>Mike</td>
<td>MacClellan</td>
<td>1945</td>
</tr>
<tr>
<td>Mabel</td>
<td>MacClellan</td>
<td>1945</td>
</tr>
<tr>
<td>Marcel</td>
<td>McClellan</td>
<td>1958</td>
</tr>
</tbody>
</table>

The % wildcard in that query stands for "zero or more unspecified characters."

**Linking Tables with Inner Joins**

Joins inspire joy in the hearts of database people the way navigational charts thrill sailors and pilots. Anyone who's read a general-interest book about relational databases is aware that joins have to do with extracting data from related tables, but it's generally only those who actually work with databases who can perform a join with any sort of alacrity. Anyone can write a single-table SELECT statement; it takes some experience to craft a join.

That said, joins aren't that complicated. They're really just slightly more complicated SELECT queries. As is the case with ordinary single-table SELECT queries, you write a query by nominating the tables and columns that interest you, and specifying the Boolean rules that determine which rows should be selected. The syntax is just slightly longer because there are more tables involved.

Imagine there is a company table that was created with this statement:

```sql
CREATE TABLE 'company'
('id' int NOT NULL AUTO_INCREMENT,
 'name' varchar(30) NOT NULL DEFAULT '',
 PRIMARY KEY ('id'))
TYPE=InnoDB;
```

and which looks like this:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AlphaCo</td>
</tr>
<tr>
<td>2</td>
<td>BoBoCo</td>
</tr>
<tr>
<td>3</td>
<td>CapCorp</td>
</tr>
</tbody>
</table>

Further suppose we have a contact table that was created with this statement:

```sql
CREATE TABLE 'contact'
('id' int NOT NULL AUTO_INCREMENT,
 'name' varchar(30) NOT NULL DEFAULT '',
 'companyID' int NOT NULL DEFAULT '',
...
PRIVMRY KEY ('id'),
INDEX (companyID),
FOREIGN KEY (companyID)
REFERENCES company(id)
ON UPDATE CASCADE ON DELETE RESTRICT)
TYPE=InnoDB;

and which looks like this:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>companyID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Camille</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Dov</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that, as discussed in Chapter 6, we're using the InnoDB table type because it provides strict enforcement of relationships.

We want a query that returns the name field from the contact table, next to the corresponding name field from the company table. First, we specify the columns that interest us, using a dot notation to be explicit about which table each column comes from:

```
SELECT contact.name AS contactName, company.name AS companyName
```

The two AS clauses ("AS contactName" and "AS companyName") are necessary because both tables have columns called name, and we need to differentiate them in the results of the query.

Then, we have to explicitly name the tables in the FROM clause:

```
FROM contacts, company
```

It is in the WHERE clause that we take advantage of the foreign key in contacts, requiring that it match the primary key column in company:

```
WHERE contacts.companyID=company.id
```

The query in its entirety looks like this:

```
SELECT contact.name AS contactName, company.name AS companyName
FROM contacts, company
WHERE contact.companyID=company.id;
```

It yields the results we want:

<table>
<thead>
<tr>
<th>contactName</th>
<th>companyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>BoBoCo</td>
</tr>
<tr>
<td>Bob</td>
<td>CapCorp</td>
</tr>
<tr>
<td>Camille</td>
<td>AlphaCo</td>
</tr>
<tr>
<td>Dov</td>
<td>CapCorp</td>
</tr>
</tbody>
</table>
The function of an inner join is to take two tables, each with a number of rows, and return the data in all columns of both tables, but only for those rows for which specified conditions are true. Typically, you'll use inner joins in conjunction with a foreign key to access related data stored in another table.

One could say that data integrity comes about when you spread data across many tables, eliminating duplication and reducing the opportunity for semantic errors in which multiple pieces of data stand for the same logical thing (e.g., Boeing Corporation versus Boeing Aircraft Corporation). Informational integrity, on the other hand, in which your database (and the system you use for accessing it) is able to quickly and automatically indicate links between related table rows, comes from effective use of joins, particularly inner joins.

Inner joins are tremendously powerful, enabling you to create virtual tables (really, result sets that come from combining multiple tables), and they're the most frequently used variety of join. More often than not, you'll use an inner join when pulling information out of your database in multi-tier applications. No row is an island, and your application will be far more powerful if you can access all related pieces of data at the same time. Note that it's appropriate to do this at the accessor level, with SQL queries, because the database server can process joins very quickly. Joins are not calculations, which should happen in the business logic layer. Joins are just slightly more complicated methods of accessing data.

The generic SQL syntax for an inner join is this:

```sql
SELECT columns
FROM table1
INNER JOIN table2
ON conditions
```

To demonstrate the behavior of inner joins with an example, let's suppose we have the same tables we worked with earlier in this chapter. One table contains information about companies, another contains information about contacts at those companies, as well as some people who aren't affiliated with a particular organization.

First, the company table, which was created with this SQL statement:

```sql
CREATE TABLE `company`
(`id` int NOT NULL AUTO_INCREMENT,
 `name` varchar(30) NOT NULL DEFAULT ",",
 PRIMARY KEY (`id`))
TYPE=InnoDB
```

holds these contents:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AlphaCo</td>
</tr>
<tr>
<td>2</td>
<td>BoBoCo</td>
</tr>
<tr>
<td>3</td>
<td>CapCorp</td>
</tr>
<tr>
<td>4</td>
<td>DingDingLtd</td>
</tr>
<tr>
<td>5</td>
<td>EEEEK Inc.</td>
</tr>
</tbody>
</table>
The other table, contact, came about as a result of this SQL statement:

```
CREATE TABLE 'contact'  
('id' int NOT NULL AUTO_INCREMENT,  
 'name' varchar(30) NOT NULL DEFAULT ",",  
 'companyID' int NULL,  
 PRIMARY KEY ('id'),  
 INDEX (companyID),  
 FOREIGN KEY (companyID)  
 REFERENCES company(id))  
TYPE=InnoDB
```

and holds these data:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>companyID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Camille</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Dov</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Edgar</td>
<td>[NULL]</td>
</tr>
<tr>
<td>6</td>
<td>Francine</td>
<td>[NULL]</td>
</tr>
<tr>
<td>7</td>
<td>Gaylord</td>
<td>[NULL]</td>
</tr>
<tr>
<td>8</td>
<td>Hongbin</td>
<td>[NULL]</td>
</tr>
</tbody>
</table>

The id column in the company table is the primary key there, and the companyID column in the contact table is an integer that refers to the id column in the owner table—it's a foreign key. The foreign key may be null, indicating that there can be contacts with no associated company.

If we wanted a result set that contained full details of all companies and their employees, we'd use an inner join query like this one:

```
SELECT company.name AS companyName, contact.name AS contactName  
FROM company  
INNER JOIN contact  
ON contact.companyID=company.id;
```

These are the results of this query:

<table>
<thead>
<tr>
<th>companyName</th>
<th>contactName</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoBoCo</td>
<td>Alice</td>
</tr>
<tr>
<td>CapCorp</td>
<td>Bob</td>
</tr>
<tr>
<td>AlphaCo</td>
<td>Camille</td>
</tr>
<tr>
<td>CapCorp</td>
<td>Dov</td>
</tr>
</tbody>
</table>

If you like, you can use AND and OR to specify multiple conditions in the ON clause. The Boolean operators work just like they do in WHERE clauses.
Speaking of WHERE clauses, isn't this INNER JOIN syntax the same as the more advanced SELECT syntax we saw earlier in this chapter? Indeed it is. This query:

```
SELECT company.name AS companyName, contact.name AS contactName
FROM company, contact
WHERE contact.companyID=company.id;
```

is functionally the same as the INNER JOIN query we just tried out.

**Linking Tables with Outer Joins**

In contrast to inner joins, outer joins exist because there's often not a one-to-one relationship between rows in different tables. Often, you'll have people in your contacts table without corresponding companies in your companies table, or customer listings in the customers table without corresponding orders in the orders table. Similarly, a pet owner might have more than one pet, and a customer might have more than one order.

Outer joins differ from inner joins in that they return all rows in one table, combined with only certain rows in another table. Depending on whether you want to return all rows in the first table specified in your query, or all rows from the second, you could choose a left outer join (otherwise known as a left join) or a right outer join (a right join), respectively.

As with any query that is meant to return all rows in any table, you should use outer joins with care. If they are run on a large table, outer joins can require a lot of processing time and generate unwieldy result sets.

Left joins consider two tables, as do all joins. A left join, though, returns all rows in the first table, and only those rows in the second table that have relational links to the first table. The generic syntax for a left join is this:

```
SELECT columns
FROM left_table
LEFT [OUTER] JOIN right_table
ON conditions
```

Consider our contacts-and-companies example again. It's completely reasonable to think that our database would contain both companies at which we have no specific contacts, and contacts that are not associated with a company. Here is the company table:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AlphaCo</td>
</tr>
<tr>
<td>2</td>
<td>BoBoCo</td>
</tr>
<tr>
<td>3</td>
<td>CapCorp</td>
</tr>
<tr>
<td>4</td>
<td>DingDingLtd</td>
</tr>
<tr>
<td>5</td>
<td>EEEEK Inc.</td>
</tr>
</tbody>
</table>
And here is the contact table:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>companyID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Camille</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Dov</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Edgar</td>
<td>[NULL]</td>
</tr>
<tr>
<td>6</td>
<td>Francine</td>
<td>[NULL]</td>
</tr>
<tr>
<td>7</td>
<td>Gaylord</td>
<td>[NULL]</td>
</tr>
<tr>
<td>8</td>
<td>Hongbin</td>
<td>[NULL]</td>
</tr>
</tbody>
</table>

Suppose we require an accessor function that returns a list of all companies in the company table, as well as contacts for those companies where they exist in the contact table. This is an appropriate application for a left join in which the company table is the left table, and the contact table is the right. Here's the query we'd use:

```
SELECT company.name AS companyName, contact.name AS contactName
FROM company
LEFT JOIN contact
ON company.id=contact.companyID;
```

The results of that query are exactly what we want:

<table>
<thead>
<tr>
<th>companyName</th>
<th>contactName</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaCo</td>
<td>Camille</td>
</tr>
<tr>
<td>BoBoCo</td>
<td>Alice</td>
</tr>
<tr>
<td>CapCorp</td>
<td>Bob</td>
</tr>
<tr>
<td>CapCorp</td>
<td>Dov</td>
</tr>
<tr>
<td>DingDingLtd</td>
<td>[NULL]</td>
</tr>
<tr>
<td>EEEEK Inc.</td>
<td>[NULL]</td>
</tr>
</tbody>
</table>

Note that Edgar, Francine, Gaylord, and Hongbin, who are in the contact table but who have no company association, appear nowhere in the result set. Also note that DingDingLtd and EEEEK Inc., which appear in the company table but have no associated contacts, are listed in the result set, but with null values in the columns where the contact information should go.

Right joins, as you might expect, are the opposite of left joins. A right join returns all rows in the second table, and only those rows in the first table that have relational links to the second table. The generic syntax for a right join is as follows:

```
SELECT columns
FROM left_table
RIGHT [OUTER] JOIN right_table
ON conditions
```
A right join, applied to our companies-and-contacts database, would return all contacts regardless of whether they had associated companies, and would return only those company rows with associated contacts. The syntax specific to this application is:

```sql
SELECT contact.name AS contactName, company.name AS companyName
FROM company
RIGHT JOIN contact
ON company.id=contact.companyID;
```

It's nearly the same as the statement we used in the left join demonstration, with the exception of the keyword RIGHT. The results, however, are significantly different:

<table>
<thead>
<tr>
<th>contactName</th>
<th>companyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>BoBoCo</td>
</tr>
<tr>
<td>Bob</td>
<td>CapCorp</td>
</tr>
<tr>
<td>Camille</td>
<td>AlphaCo</td>
</tr>
<tr>
<td>Dov</td>
<td>CapCorp</td>
</tr>
<tr>
<td>Edgar</td>
<td>[NULL]</td>
</tr>
<tr>
<td>Francine</td>
<td>[NULL]</td>
</tr>
<tr>
<td>Gaylord</td>
<td>[NULL]</td>
</tr>
<tr>
<td>Hongbin</td>
<td>[NULL]</td>
</tr>
</tbody>
</table>

Note that null values again appear in the company.name column for contacts that do not have associated company rows. Note also (again) that the contact table was listed in its entirety—a potentially risky phenomenon in situations in which the table contained many rows.

### 7.1.2 SELECT Queries for Currawong Accounting

In our multicurrency accounting application, we need a number of accessor-layer programs based on SELECT queries. Specifically, we need classes that will do the following tasks:

- Return all payees,
- Return all bank accounts,
- Return all currencies,
- Return all institutions,
- Return all accounts,
- Return all account types,
- Return all transactions,
- Return an account for which an id value is specified,
- Return an account type for which an id value is specified,
Return a bank account for which an id value is specified,
Return a currency for which an id value is specified,
Return an institution for which an id value is specified,
Return a payee for which an id value is specified,
Return a transaction for which an id value is specified,
Return a transaction type for which an id value is specified,
Return the balance of a bank account on a specific date, given the account's id value and the date,
Return the average balance of an account during a specified number of days prior to a specified date, given the account's id value, the date, and a number of days (this will be used for moving-average calculations), and
Return a range of transactions from a specified account, given the account's id value, a start date, and an end date.

Most of these require only a simple SELECT statement, but others require that joins be used. The last eight items in this list are functionally very similar. They're all dealt with in the section titled, "Return a Specified Row from a Specified Table." Let's examine solutions to each of the requirements in turn.

**Return All Payees**

To return all payees, we obviously need a SELECT statement that draws all columns out of the ACCT_payee table. Because ACCT_payee has no foreign keys, extracting its data is a simple matter of sending a straightforward SELECT statement to the database server via a PEAR DB connection. Let's examine getPayees.php line by line to see how this is done.

```php
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getPayees');
$s->service($HTTP_RAW_POST_DATA);
```

The first order of business is to import the PEAR DB classes, the NuSOAP classes, and dbDetails.php, which contain information about the database server and security credentials for it. Then, we register the getPayees function—to be defined momentarily—as a SOAP service. Refer to Chapters 4 and 5 for further information on NuSOAP and Web services.
function getPayees() {

    // Make globals from configuration.php available within function scope.
    global $phptype;    // Type of database server.
    global $hostspec;   // Hostname of database server.
    global $database;   // Database name.
    global $username;   // Database username.
    global $password;   // Database password.

    In the function definition, these five global variables must be declared in order for them
    to be accessible. Their values are assigned in dbDetails.php, which was imported at the
    beginning of the program.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...
    $dsn = "$phptype://$username:$password@$hostspec/$database";

    $db = DB::connect($dsn);

    if (DB::isError($db)) {
        die ($db->getMessage());
    }

    Using the standard PEAR DB procedure (see Chapter 5 for more details), we connect to the
    database server. The program checks for an error condition.

    // Assemble and send SQL statement, allowing for errors...

    $sql = "select id, name, streetAddress, city, state, postcode, country
     from ACCT_payee";

    That's the SQL query that is to be sent to the database server. Note that we specify the
    columns, even though it's all of them. That way we know what order the columns will be
    in when results come back.

    $result = $db->query($sql);

    if (DB::isError($result))
    {
        $errorMessage = $result->getMessage();
        die ($errorMessage);
    }

    The program sends the query to the database and checks to see if an error message comes
    back.

    $returnArray = array();
// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
    $id = $row[0];
    $name = $row[1];
    $streetAddress = $row[2];
    $city = $row[3];
    $state = $row[4];
    $postcode = $row[5];
    $country = $row[6];

    $returnArray[] = array('id' => $id, 'name' => $name, 'streetAddress' => $streetAddress, 'city' => $city, 'state' => $state, 'postcode' => $postcode, 'country' => $country);
}

// Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

The remainder of the program involves setting up an array—$returnArray—and filling it with a series of subarrays (in other words, $returnArray is a two-dimensional array). The subarrays are associative arrays in which the keys correspond to database column names and values come from each row of results that comes out of $result by way of fetchRow().

### Return All Bank Accounts

In order to return a list of all bank accounts, the database server will need to perform several joins. This is because the ACCT_bank_account table, which contains information on individual accounts maintained by the customer, has a number of foreign keys. Other than the complex SQL query, though, the accessor-layer program that retrieves account information is similar to other accessor-layer programs.

Here's a listing of getBankAccounts.php, which handles account-list retrieval.

```php
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
```
// and register function as Web Service...

$s = new soap_server;
$s->register('getBankAccounts');
$s->service($HTTP_RAW_POST_DATA);

function getBankAccounts() {

    // Make globals from configuration.php available within function scope.
    global $phptype;  // Type of database server.
    global $hostspec;  // Hostname of database server.
    global $database;  // Database name.
    global $username;  // Database username.
    global $password;  // Database password.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...

    $dsn = "$phptype://$username:$password@$hostspec/$database";

    $db = DB::connect($dsn);

    if (DB::isError($db)) {
        die ($db->getMessage());
    }

    // Assemble and send SQL statement, allowing for errors...

    $sql = "<<EOQ
SELECT
    ACCT_bank_account.id,
    ACCT_institution.institutionName AS institution,
    ACCT_bank_account.number,
    ACCT_currency.abbreviation AS currency,
    ACCT_currency.id AS currencyld,
    ACCT_acct_type.name AS type,
    ACCT_bank_account.description
FROM ACCT_bank_account
LEFT JOIN ACCT_currency
ON ACCT_bank_account.currency=ACCT_currency.id
LEFT JOIN ACCT_acct_type
ON ACCT_acct_type.id=ACCT_bank_account.type
LEFT JOIN ACCT_institution
ON ACCT_institution.id=ACCT_bank_account.institution";

$result = $db->query($sql);

if (DB::isError($result))
{
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

$returnArray = array();
// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.
while ($row = $result->fetchRow())
{
    $id = $row[0];
    $institution = $row[1];
    $number = $row[2];
    $currency = $row[3];
    $currencyId = $row[4];
    $type = $row[5];
    $description = $row[6];
    $returnArray[] = array('id' => $id, 'institution' => $institution,
        'number' => $number, 'currency' => $currency, 'currencyId' => $currencyId,
        'type' => $type, 'description' => $description);
}

// Disconnect from database and return $returnArray...
$db->disconnect();
return $returnArray;


Again, this program bears a strong resemblance to other programs in the accessor layer, with the exception of this complicated SQL statement:

SELECT
    ACCT_bank_account.id,
    ACCT_institution.institutionName AS institution,
    ACCT_bank_account.number,
Chapter 7: The Accessor Layer

ACCT_currency.abbreviation AS currency,
ACCT_currency.id AS currencyId,
ACCT_acct_type.name AS type,
ACCT_bank_account.description
FROM ACCT_bank_account
LEFT JOIN ACCT_currency
ON ACCT_bank_account.currency=ACCT_currency.id
LEFT JOIN ACCT_acct_type
ON ACCT_acct_type.id=ACCT_bank_account.type
LEFT JOIN ACCT_institution
ON ACCT_institution.id=ACCT_bank_account.institution

The statement specifies that seven columns are to be retrieved from four different tables. Not surprisingly, ACCT_bank_account is the main table, and the other three tables are left joined to it based on foreign key values in ACCT_bank_account. Performing multiple joins in a single query is not a problem. Note also that a number of retrieved columns are renamed with AS statements.

Return All Currencies

The requirement to return a list of all currencies stored in the database is handled by the code contained in getCurrencies.php. It's a straightforward query module with no joins or other unusual tricks. Here's a full listing:

```php
require_once('nusoap-0.6/nusoap.php') ;
require_once('configuration.php') ;
// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getCurrencies');
$s->service($HTTP_RAW_POST_DATA);

function getCurrencies() {
  // Make globals from configuration.php available within function scope.
  global $phptype;  // Type of database server.
  global $hostspec;  // Hostname of database server.
  global $database;  // Database name.
  global $username;  // Database username.
  global $password;  // Database password.

  // Assemble Data Source Name (DSN) and connect, allowing for errors...
```
$dsn = "$phptype://$username:$password@$hostspec/$database";

$db = DB::connect($dsn);

if (DB::isError($db)) {
    die ($db->getMessage());
}

// Assemble and send SQL statement, allowing for errors...

$sql = "<<EOQ
select id, abbreviation, country, name, xRate, DATE_FORMAT(updated, '%d %M %Y %T')
from ACCT_currency
EOQ;

// Note that the MySQL DATE_FORMAT() function is used to get the string into
the required format.

$result = $db->query($sql);

if (DB::isError($result))
{
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
    $id = $row[0];
    $abbreviation = $row[1];
    $country = $row[2];
    $name = $row[3];
    $xRate = $row[4];
    $updated = $row[5];
$returnArray[] = array('id' => $id, 'abbreviation' => $abbreviation, 'country' => $country, 'name' => $name, 'xRate' => $xRate, 'updated' => $updated);
}

// Disconnect from database and return $returnArray...
$db->disconnect();

return $returnArray;
}

Simple enough. Again, the returned array—$returnArray—is a two-dimensional array in which the subarrays are associative.

**Return All Institutions**

Currawong Accounting must be able to return a list of all financial institutions with which the customer does business. Because the table that stores information about financial institutions, ACCT_institutions, has no foreign keys, there are no joins to be set up. The accessor program simply connects via PEAR DB, runs a simple query, and returns the results in an array. Here is a full listing of getInstitutions.php:

```php
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getInstitutions');
$s->service($HTTP_RAW_POST_DATA);

function getInstitutions() {

// Make globals from configuration.php available within function scope.
global $phptype;    // Type of database server.  
global $hostspec;  // Hostname of database server.  
global $database;  // Database name.  
global $username;  // Database username.  
global $password;  // Database password.  

// Assemble Data Source Name (DSN) and connect, allowing for errors...

$dsn = "$phptype://$username:$password@$hostspec/$database";
```
$db = DB::connect($dsn);

if (DB::isError($db)) {
    die ($db->getMessage());
}

// Assemble and send SQL statement, allowing for errors...
$sql = "select id, institutionName AS name, streetAddress, city, state, postcode, country from ACCT_institution";
$result = $db->query($sql);
if (DB::isError($result))
    {
        $errorMessage = $result->getMessage();
        die ($errorMessage);
    }
$returnArray = array();

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.
while ($row = $result->fetchRow())
    {
        $id = $row[0];
        $name = $row[1];
        $streetAddress = $row[2];
        $city = $row[3];
        $state = $row[4];
        $postcode = $row[5];
        $country = $row[6];
        $returnArray[] = array('id' => $id, 'name' => $name, 'streetAddress' => $streetAddress, 'city' => $city, 'state' => $state, 'postcode' => $postcode, 'country' => $country);
    }

// Disconnect from database and return $returnArray...
$db->disconnect();
This is an accessor program like many others for Currawong Accounting. The SQL statement is simple, and the results are made available via a Web service in the form of an array of associative arrays.

**Return All Accounts**

Currawong Accounting must be able to return the names of all accounts—that is to say, all "accounting accounts," not all bank accounts in this case. This is a very simple operation, requiring only an uncomplicated SELECT query and the usual NuSOAP and array-returning infrastructure. Here is a full listing of `getAccounts.php`:

```php
require_once( 'nusoap-0.6/nusoap.php' );
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getAccounts');
$s->service($HTTP_RAW_POST_DATA);

function getAccounts() {

    // Make globals from configuration.php available within function scope.
    global $phptype;    // Type of database server.
    global $hostspec;   // Hostname of database server.
    global $database;   // Database name.
    global $username;   // Database username.
    global $password;   // Database password.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...

    $dsn = "$phptype://$username:$password@$hostspec/$database";
    $db = DB::connect($dsn);

    if (DB::isError($db)) {
        die ($db->getMessage());
    }
}
// Assemble and send SQL statement, allowing for errors...

$sql = "select id, name from ACCT_account";

$result = $db->query($sql);

if (DB::isError($result))
{
$errorMessage = $result->getMessage();
die ($errorMessage);
}

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
$id = $row[0];
$name = $row[1];
$returnArray[] = array('id' => $id, 'name' => $name);
}

// Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

} Again, quite simple. The program runs an unadorned SELECT statement:

SELECT id, name from ACCT_account

and returns the resultant rows in an indexed array of associative arrays.

Return All Account Types

The requirement to return a list of all account types stored in the database is handled by
the code contained in getAcctTypes.php. It's a straightforward query module. Here's a full
listing:

getAcctTypes.php

require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getAcctTypes');
$s->service($HTTP_RAW_POST_DATA);

function getAcctTypes() {

    // Make globals from configuration.php available within function scope.
    global $phptype;    // Type of database server.
    global $hostspec;   // Hostname of database server.
    global $database;   // Database name.
    global $username;   // Database username.
    global $password;   // Database password.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...
    $dsn = "$phptype://$username:$password@$hostspec/$database";
    $db = DB::connect($dsn);
    if (DB::isError($db)) {
        die ($db->getMessage());
    }

    // Assemble and send SQL statement, allowing for errors...
    $sql = "select id, name from ACCT_acct_type";
    $result = $db->query($sql);
    if (DB::isError($result))
    {
        $errorMessage = $result->getMessage();
        die ($errorMessage);
    }

    $returnArray = array()

    // Extract rows from query results, fitting pieces of data into
    // $returnArray (an associative array) for returning.
while ($row = $result->fetchRow())
{
    $id = $row[0];
    $name = $row[1];

    $returnArray[] = array('id' => $id, 'name' => $name);
}

// Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

}

The core of this module is a simple SELECT statement that returns two columns, id and name, from every row in the table.

Return All Transactions

The program is required to return all transactions stored in the ACCT_register table. Although the requirement is easily satisfied with a SELECT query, the query must include a couple of JOIN operations because ACCT_register includes several foreign keys. Here is the complete listing of getTransactions.php, the file that handles the work:

require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getTransactions');
$s->service($HTTP_RAW_POST_DATA);

function getTransactions() {

    // Make globals from configuration.php available within function scope.
    global $phptype; // Type of database server.
    global $hostspec; // Hostname of database server.
    global $database; // Database name.
    global $username; // Database username.
global $password; // Database password.

// Assemble Data Source Name (DSN) and connect, allowing for errors...

$dsn = "$phptype://$username:$password@$hostspec/$database";

$db = DB::connect($dsn);

if (DB::isError($db)) {
    die ($db->getMessage());
}

// Assemble and send SQL statement, allowing for errors...

$sql = "SELECT
ACCT_register.id,
ACCT_register.date,
ACCT_register.number,
ACCT_payee.name AS payee,
ACCT_register.amount,
ACCT_account.name AS account
FROM ACCT_register
LEFT JOIN ACCT_payee
ON ACCT_register.payee=ACCT_payee.id
LEFT JOIN ACCT_account
ON ACCT_register.account=ACCT_account.id
EOQ;"

$result = $db->query($sql);

if (DB::isError($result)) {
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

$returnArray = array();

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
The unusual aspect of this program is the elaborate SELECT statement, which involves two left joins:

```sql
SELECT
    ACCT_register.id,
    ACCT_register.date,
    ACCT_register.number,
    ACCT_payee.name AS payee,
    ACCT_register.amount,
    ACCT_account.name AS account
FROM ACCT_register
LEFT JOIN ACCT_payee
    ON ACCT_register.payee=ACCT_payee.id
LEFT JOIN ACCT_account
    ON ACCT_register.account=ACCT_account.id
```

The two joins correspond to the two foreign keys in ACCT_register, which have to do with the payee and the accounting account involved in the transaction.

**Return a Specified Row from a Specified Table**

The Currawong specification requires a number of accessor-layer programs for similar purposes. These all take the primary key value (id) of some row in some table and return
all or most column values from that row. Here’s a list of all accessor-layer programs that do that kind of work:

- Return a specified account (getSpecifiedAccount.php)
- Return a specified account type (getSpecifiedAcctType.php)
- Return a specified bank account (getSpecifiedBankAccount.php)
- Return a specified currency (getSpecifiedCurrency.php)
- Return a specified institution (getSpecifiedInstitution.php)
- Return a specified payee (getSpecifiedPayee.php)
- Return a specified transaction (getSpecifiedTransaction.php)
- Return a specified transaction type (getSpecifiedTransType.php)

All of these programs are essentially the same, differing only in the tables against which their queries run. Let’s have a look at getSpecifiedBankAccount.php, which can exemplify the bunch:

```php
getSpecifiedBankAccount.php

require_once('DB.php');
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getSpecifiedBankAccount');
$s->service($HTTP_RAW_POST_DATA);

function getSpecifiedBankAccount($id) {

    // Make globals from configuration.php available within function scope.
    global $phptype;  // Type of database server.
    global $hostspec; // Hostname of database server.
    global $database; // Database name.
    global $username; // Database username.
    global $password; // Database password.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...

    $dsn = "$phptype://$username:$password@$hostspec/$database";
    $db = DB::connect($dsn);
```
if (DB::isError($db)) {
    die ($db->getMessage());
}

// Assemble and send SQL statement, allowing for errors...

$sql = <<<EOQ
SELECT
    ACCT_bank_account.id,
    ACCT_institution.institutionName AS institution,
    ACCT_institution.id AS institutionId,
    ACCT_bank_account.number,
    ACCT_currency.abbreviation AS currency,
    ACCT_currency.id AS currencyId,
    ACCT_acct_type.name AS type,
    ACCT_acct_type.id AS acctTypeId,
    ACCT_bank_account.description
FROM ACCT_bank_account
LEFT JOIN ACCT_currency
    ON ACCT_bank_account.currency=ACCT_currency.id
LEFT JOIN ACCT_acct_type
    ON ACCT_acct_type.id=ACCT_bank_account.type
LEFT JOIN ACCT_institution
    ON ACCT_institution.id=ACCT_bank_account.institution
WHERE
    ACCT_bank_account.id="$id"
EOQ;

$result = $db->query($sql);

if (DB::isError($result)) {
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

$returnArray = array();

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
Essentially, that program makes the same query against ACCT_bank_account as getBankAccounts.php (which returns all bank accounts), except that this program contains a WHERE clause:

```sql
WHERE
ACCT_bank_account.id="$id"
```

Because $id arrived as a parameter, it can be used to grab the details of only the account required by the remote tier.

**Return the Balance of an Account**

The accessor layer has to be able to return the balance of a bank account on a specific date, given the account's id value and the date. This job is handled by getSpecifiedBankAccountBalance.php, listed here:

```
getSpecifiedBankAccountBalance.php
```

```php
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');
```
// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getSpecifiedBankAccountBalance');
$s->service($HTTP_RAW_POST_DATA);

function getSpecifiedBankAccountBalance($id, $date) {

    // Make globals from configuration.php available within function scope.
    global $phptype; // Type of database server.
    global $hostspec; // Hostname of database server.
    global $database; // Database name.
    global $username; // Database username.
    global $password; // Database password.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...
    $dsn = "$phptype://$username:$password@$hostspec/$database";
    $db = DB::connect($dsn);

    if (DB::isError($db)) {
        die ($db->getMessage());
    }

    // Assemble and send SQL statement, allowing for errors...

    $sql = <<<EOQ
SELECT
    sum(amount) as balance
FROM
    acct_register
WHERE
    bank_account="$id"
AND
    date < "$date"
EOQ;

$result = $db->query($sql);

if (DB::isError($result)) {
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}
$returnArray = array();

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
    $balance = $row[0];

    $returnArray[] = array('balance' => $balance);
}

// Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

That program accepts the id (primary key value) of the account to be examined as a parameter, as well as the date for which the balance is to be calculated. The date value has to be in MySQL format (YYYY-MM-DD) when it arrives as a parameter; no verification of its format is done or conversion made.

The id and date values are used in a query:

```
SELECT
    sum(amount) as balance
FROM
    acct_register
WHERE
    bank_account="$id"
    AND
    date < "$date"
```

There, sum() is a MySQL function that totals all values in a specified column (amount, in this case). The query is further limited by the WHERE clause, which imposes limits based on both id and date.

**Return the Average Balance of an Account over Time**

Currawong Accounting needs a module on the accessor layer that will return the average balance of an account during a specified number of days prior to a specified date, given
the account's id value, the date, and a number of days. The idea is that the user can specify
an account, a date, and a number of days (typically 60 or 90) and see the average balance
(based on weekly samples) for that account during the preceding number of days specified.
For example, if the user specified account 2 (that's an id value), today's date, and the value
60, this module should calculate the mean balance of account 2 during the past 60 days.
This work is carried out by getAverageBalance.php.

getAverageBalance.php

require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');
require_once('dateConv.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getAverageBalance');
$s->service($HTTP_RAW_POST_DATA);

function getAverageBalance($account, $date, $days) {
    // $date comes in MySQL YYYY-MM-DD format
    // Make globals from configuration.php available within function scope.
    global $phptype; // Type of database server.
    global $hostspec; // Hostname of database server.
    global $database; // Database name.
    global $username; // Database username.
    global $password; // Database password.

    $date = mysql_datetime_to_timestamp($date);
    $midDate = $date - ($days * 24 * 60 * 60);

    $total = 0;
    $n = 0;

    // Assemble Data Source Name (DSN) and connect, allowing for errors...

    $dsn = "$phptype://$username:$password@$hostspec/$database";
    $db = DB::connect($dsn);

    if (DB::isError($db))
    {
        die ($db->getMessage());
    }
while ($midDate < $date)
{

$midDate = timestamp_to_mysql_date($midDate);

// Assemble and send SQL statement, allowing for errors...

$sql = "SELECT sum(amount) as balance FROM acct_register WHERE bank_account="$account" AND date < "$midDate"
EOQ;
$result = $db->query($sql);

if (DB::isError($result))
{
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

// Extract rows from query results, fitting pieces of data into $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
    $balance = $row[0];
    $total += $balance;
}

$n++;
$midDate = mysql_datetime_to_timestamp($midDate);
$midDate += 604800;

$movingAverage = $total / $n;

// Disconnect from database and return $returnArray...
Note that getAverageBalance.php makes use of an unusual feature of PHP syntax in enclosing its SQL statement. The following sequence of characters is the functional equivalent of a double quotation mark ("') in the opening position (that is, at the beginning of a quoted string).

```php
<<<EOQ
```

Its complement is this:

```php
EOQ;
```

That is the equivalent of a double quotation mark ("') at the end of a quoted string. Between the two sequences you can have as many line breaks as you like; the PHP interpreter ignores them. The EOQ sequences allow you to break up long strings and thus make your source code more readable.

This program works by taking the provided date value, converting it to a Unix-standard timestamp, and subtracting from it a value equal to the number of seconds in the period for which an average value is required:

```
$date = mysql_datetime_to_timestamp($date);
$midDate = $date - ($days * 24 * 60 * 60);
```

Then, a loop gets the balance at the end of every seven-day period between that calculated starting point and the original date value. On each pass through the loop, the balance is added to a running total, and $n is incremented so the total number of samples is known:

```
while ($midDate < $date)
{
    $midDate = timestamp_to_mysql_date($midDate);

    $sql = <<<EOQ
    SELECT
    sum(amount) as balance
    FROM
    acct_register
    WHERE
    bank_account="$account"
    AND
    date < "$midDate"
EOQ;

    $result = $db->query($sql);
```
if (DB::isError($result))
{
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

while ($row = $result->fetchRow())
{
    $balance = $row[0];
    $total += $balance;
}

$n++;
$midDate = mysql_datetime_to_timestamp($midDate);
$midDate += 604800;

}

After the loop terminates, the average of all the samples is computed:

$movingAverage = $total / $n;

and returned:

return $movingAverage;

Return a Range of Transactions

In order for the presentation layer to be able to depict an account register, it has to be able to retrieve a series of transactions from an account of interest. The module contained in getSpecifiedTransactions.php will return a range of transactions from a specified account, given the account's id value, a start date, and an end date.

getSpecifiedTransactions.php

require_once('DB.php');
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getSpecifiedTransactions');
$s->service($HTTP_RAW_POST_DATA);

function getSpecifiedTransactions($startDate, $endDate, $account) {

// Make globals from configuration.php available within function scope.
global $phptype;    // Type of database server.
global $hostname;  // Hostname of database server.
global $database;  // Database name.
global $username;  // Database username.
global $password;  // Database password.

// Assemble Data Source Name (DSN) and connect, allowing for errors...

$dsn = "$phptype://$username:$password@$hostname/$database";

$db = DB::connect($dsn);

if (DB::isError($db)) {
    die ($db->getMessage());
}

// Assemble and send SQL statement, allowing for errors...

$sql = <<<EOQ
SELECT
    ACCT_register.id,
    ACCT_register.date,
    ACCT_register.number,
    ACCT_register.memo,
    ACCT_payee.name AS payee,
    ACCT_register.amount,
    ACCT_account.name AS account,
    ACCT_trans_type.name AS type
FROM ACCT_register
LEFT JOIN ACCT_payee
ON ACCT_register.payee=ACCT_payee.id
LEFT JOIN ACCT_account
ON ACCT_register.account=ACCT_account.id
LEFT JOIN ACCT_trans_type
ON ACCT_register.type=ACCT_trans_type.id
WHERE
    ACCT_register.date
BETWEEN
    "$startDate" AND "$endDate"
AND
    ACCT_register.bank_account="$account"
EOQ;

$result = $db->query($sql);
if (DB::isError($result))
{
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

$returnArray = array();

// Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
    $id = $row[0];
    $date = $row[1];
    $number = $row[2];
    $memo = $row[3];
    $payee = $row[4];
    $amount = $row[5];
    $account = $row[6];
    $type = $row[7];

    $returnArray[] = array('id' => $id, 'date' => $date, 'number' => $number,
                            'memo' => $memo, 'payee' => $payee, 'amount' => $amount,
                            'account' => $account, 'type' => $type);
}

// Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

This is not a complicated program. Its SQL query simply retrieves all fields in the
ACCT_register table that have the right account value and fit within the date parameters:

SELECT
    ACCT_register.id,
    ACCT_register.date,
    ACCT_register.number,
ACCT_register.memo,
ACCT_payee.name AS payee,
ACCT_register.amount,
ACCT_account.name AS account,
ACCT_trans_type.name AS type
FROM ACCT_register
LEFT JOIN ACCT_payee
ON ACCT_register.payee=ACCT_payee.id
LEFT JOIN ACCT_account
ON ACCT_register.account=ACCT_account.id
LEFT JOIN ACCT_trans_type
ON ACCT_register.type=ACCT_trans_type.id
WHERE
ACCT_register.date
BETWEEN
"$startDate" AND "$endDate"
AND
ACCT_register.bank_account="$account"

As is typical of the software in this layer, the retrieved values are then fitted into a two-dimensional associative array that's returned to the sender.

7.2 Adding Data to the Database

Another important function of the database is to take additions—to store information. Currawong Accounting, although not expected to handle a high volume of transactions, will need to store information about transactions, as well as information about new payees, currencies, accounts, and other aspects of accounting that will be added over time.

Generally speaking, adding data to databases is accomplished through the use of SQL INSERT queries (as well as UPDATE and other queries). This section deals with INSERT queries as both pure SQL and as part of the accessor layer of our imaginary client's multicurrency accounting application.

7.2.1 Using INSERT Queries

You should use INSERT queries to add new rows to the database. The simplest INSERT query presumes knowledge of the column headers and their sequence in the table. Assume that, at the starting point, the demo table contains this:

<table>
<thead>
<tr>
<th>id</th>
<th>lastName</th>
<th>firstName</th>
<th>yearOfBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Baldoni</td>
<td>Burt</td>
<td>1956</td>
</tr>
<tr>
<td>2</td>
<td>Carey</td>
<td>Casey</td>
<td>1982</td>
</tr>
<tr>
<td>3</td>
<td>Delaney</td>
<td>Dave</td>
<td>1977</td>
</tr>
</tbody>
</table>