Careful design and construction of your multi-tier application's back end is vital to the success of the application as a whole. If you carefully design the database that persistently stores your application's information, you'll have an easier time building the rest of its software modules. This chapter aims to show you how to design and set up your application's back end.

6.1 Choosing a Persistent Storage Mechanism

Technically, your database can be any mechanism capable of storing data when the power goes off. You can use files for this purpose, but lacking a compelling reason to use them, you'll want to use a Relational Database Management System (RDBMS). RDBMSs, we'll just call them database servers, because that's what everyone calls them in real life, have lots of advantages over file-based storage, including:

- Easy storage and retrieval of data. All significant database servers support standard structured query language (SQL), as well as (usually) a number of proprietary and semiproprietary extensions to the basic language.
- Optimization. Through a variety of generally opaque magic, database servers do their best to speed the process of data storage and retrieval.
- The ability to recover from design and implementation errors. Most database servers have a journaling feature that allows you to "turn back the clock" on a database, undoing all changes that occurred since a particular time. This is handy when you make an error in the software that communicates with the database, and it makes unwanted changes to the stored data. They also support the concept of transactions,
which are sets of related changes. For example, in a bank, a transfer transaction comprises a reduction in the value of one account and an increase in the value of another. You can't have one without the other, without having a real problem.

- Security. Database servers manage user access rights, either by examining users' rights on an operating system or by keeping track of their own user lists and rights assignments.
- Survivability. Database servers have support for data redundancy and backup operations that help guarantee that important data can survive disasters.

For those reasons, the persistent storage in a multi-tier software system is usually handled by a database management server.

### 6.1.1 Choosing a Database Server

Because we're going to write our accessor layer, the layer of PHP software that directly interacts with the database server, around the PEAR DB library, we're limited (unless we want to write a custom PEAR DB extension) to the servers with which PEAR DB interfaces. Fortunately, the list is long, and includes all the major open-source and commercial database servers:

- MySQL (http://www.mysql.com)
- PostgreSQL (http://www.postgresql.com)
- InterBase (http://www.borland.com/interbase)
- Mini SQL (mSQL) (http://www.hughes.com.au)
- Microsoft SQL Server (http://www.microsoft.com/sql)
- Oracle 7/8/8i (http://www.oracle.com)
- Open Database Connectivity (ODBC) (http://msdn.microsoft.com)
- SyBase (http://www.sybase.com)
- Informix (http://www-3.ibm.com/software/data/informix)
- FrontBase (http://www.frontbase.com)

The only major database server that's missing is IBM DB2, and non-PEAR libraries exist to allow connectivity to that server. Further, note that ODBC gives us a way to access other databases for which there is an ODBC driver. ODBC adds yet another layer of abstraction (and hence some processing overhead), and it's only relevant in the Microsoft Windows environment, but it's an option.

In this book, we'll use MySQL 4.01. It's free for our purposes, and, with the InnoDB table type that's discussed in the next section, it provides all the features we require. Its performance is good, too—far more than adequate for anything we need here. MySQL is available for download from MySQL AB of Germany.
6.1.2 Selecting Development Tools

Although it's almost always possible to manipulate your database server using only the tools with which it ships, MySQL, for example, has an extensive command-line interface. You'll often benefit by having other tools on hand. These tend to be specialized in purpose, and so make specific parts of the design and implementation process faster and easier. In the case of MySQL, a whole range of excellent tools has sprung up. Many of them are open source and free to use under most circumstances; others are commercial but available in a trial version that has limited capabilities or stops working after a specified time.

On the other hand, you should remember that MySQL has its own command-line interface (documented at http://www.mysql.com). Pretty much every MySQL task can be done with the command-line interface, albeit often with a lot of hassle.

MySQL Control Center

MySQL AB, the same company that puts out MySQL itself, publishes MySQL Control Center (MySQLCC). It's a client that connects to MySQL database servers and the databases that run on them. Though it's still early in its release cycle, not yet at version 1.0 as of publication, and once in a while exhibits some odd behavior (like not telling you there's a problem when you ask it to make an InnODB table, and the server won't let it) it's still hugely useful. It's at http://www.mysql.com.

CASE Studio

A product of Charonware, CASE studio is a database design and documentation tool. Basically, it runs under Windows and hooks up to a database—it supports several database servers—over an ODBC connection. Over that connection, it can reverse engineer an existing database, effectively figuring out all of its table relationships for purposes of documentation or design validation. If you register your copy of CASE Studio, you can work the other way and have the software build a database based on your schematic diagram. You'll find CASE Studio at http://www.casestudio.com.

VMWare

When designing a multi-tier software application, it's not really possible to test the network performance of a system unless you have a number of machines connected on a local area network (LAN). Plus, you'll sometimes want to try different client/server combinations, such as connecting to a Linux-based MySQL server with a copy of MySQL Control Center that's running under Windows. The most economical way to do this is with VMWare, a product that allows you to run several completely independent virtual machines on the same physical hardware. You need a pretty hefty allotment of memory and processor grunt if you want to run more than a couple of virtual machines, but the flexibility of virtual machines is fantastic. Networking among them is easy, and it's easy to start over if you trash a configuration. VMWare can be had in a 30-day demo version at http://www.vmware.com.
6.2 Designing the Database

6.2.1 General Database Design Principles

Take time in designing your database. It is the foundation of your entire application, and time spent doing it right will pay dividends as you design and implement other layers (notably the accessor layer, which refers directly to the database). If your project (or budget) isn't big enough to farm the whole database design and implementation process off to a specialist, then allow yourself to study the problem and try a couple of alternate solutions on your own. As Frederick P. Brooks writes in *The Mythical Man-Month* (Addison-Wesley, Boston, 1991), software projects very often require an extensive false start and then a fresh beginning in order to get things right.

A number of excellent database design books exist. One of those that's most accessible is *Database Design for Mere Mortals* (Addison-Wesley, Boston, 2003), by Michael Hernandez.

The database is for data storage. Period. That principle means that the database should be a structure in which data is stored, and nothing more. No processing should take place in the database layer. Stored procedures are most definitely out. Only the storage of the data, and processes independent of the business problem (such as optimization algorithms and security-management operations) should take place in the database.

6.2.2 Specific Design Requirements

The first step in building a database is to clearly state what the database is meant to do. Consider the scope of the application your persistence layer will serve.

The design of our database has to do with the application that will be built throughout the course of this book. The nature of an application to be built would normally be established through some sort of interview process and requirements-gathering procedure (it would if you were lucky, anyway, and enjoyed the luxury of some time for gathering requirements). Here, the mission statement is just going to appear out of the void, ex nihilo, because this is a book. Here it is.

**The Mission Statement**

Our customer is an individual who provides technical consulting services to a number of customers around the world. As a result of his work, he bills his customers in a number of currencies, and maintains bank accounts in a number of financial institutions in various countries. He receives payments into and makes disbursements out of each of these accounts (though without great transaction volume), and makes transfers among them.

Our customer requires a software solution that, broadly speaking, has capabilities parallel to those of an ordinary personal-finance program, such as Quicken or Microsoft Money. The customer needs to be able to see transaction histories for each of his accounts, record new transactions, and see reports on, for example, the amount paid to a particular
vendor during a specified period. He should be able to add new accounts (including accounts at new institutions, possibly denominated in currencies previously unused in his accounting).

Unlike Quicken and Microsoft Money, though, our customer requires that his solution be able to distinguish among a large number of currencies, and be able to present reports that summarize activities across all accounts in terms of any selected currency. All currency conversions should be performed automatically, with information taken from public resources on the Internet.

Furthermore, the customer wants a single, central data store, to be run on the secure and carefully managed servers of a hosting service. Because the hosting service can most easily provide MySQL servers, that software will likely be the best choice for use on the back end. The rest of the application software should run on the hosting service's computers, as well. The customer should be able to access, via a once-per-session login procedure, all features of the application from any modern Internet browser.

Analyzing the Mission Statement

From the mission statement, we can derive a statement of what our database has to do. Let's begin with a broad statement:

The database for this application will centrally store data related to bank accounts, currencies, and transactions as specified and recorded by our customer (Figure 6.1).

Thinking about that for a second, it becomes clear that a concept such as "bank account" is complicated and potentially depends on a lot of information, like the institution that manages the account. It's important to use care as you go about setting up related tables to store application data.

6.3 Understanding Table Relationships and Normalization

Normalization is an important database concept, and one that we hear a lot about as we explore database design. Essentially, normalization is the application of principles, mostly common-sense ones, and mostly ones you're already aware of, that make relational databases more efficient.

There are six degrees of normalization, referred to as first through fifth normal form and Boyce-Codd Normal Form. First normal form can be abbreviated 1NF, second normal form as 2NF, and so on; Boyce-Codd Normal Form is shortened to BCNF.

Higher normal forms are supersets of lower ones, by definition. A database in 2NF is also in 1NF, and a database in 3NF is also in 2NF and 1NF. Normalization of a database, or examination of a database to determine its degree of normalization, is therefore an iterative process that begins with 1NF and moves up from there. Unless you're dealing with very complicated databases with elaborate relationships among tables, you can safely ignore all but 1NF.
6.3.1 First Normal Form

A table in 1NF meets two requirements.

First, it has columns that contain only atomic values, meaning values that cannot be divided meaningfully. This is why we usually split a person's name into two or three columns (firstName, surname, etc.). A single name column, which might contain something like "Benjamin Franklin," would prove a problem if we wanted to sort rows by family name (even more so if the table also contained names in which the family name came first). There are debatable points here. Is a local telephone number meaningful without its area code? Without its country code? It's up to the architect to decide whether a piece of data is atomic or requires further division.

The second requirement of a table in 1NF is that it have no repeating groups. A repeating group is any collection of two or more columns that are logically related to one another. Repeating groups often crop up when you have one-to-many and many-to-many relationships between tables. For example, if we had a company table that included three columns
representing links to the contact table, meaning each company could have up to three associated contacts, the table would have a repeating group and the table would not be in 1NF. The following table is not in 1NF.

Table: company  
name  contact1  contact2  contact3

Two of the most obvious problems with this table are that it uses three times as much space as needed for companies with one contact, and there is no neat way to associate more than three contacts with one company. It's a poor design.

The solution to the problem of repeating groups is, as is often the case, more tables. A mapping table establishes many-to-many relationships between the company table and the contact table.

Table: company  
id  name  
1  AlphaCo  
2  BoBoCo  
3  CapCorp  

Table: contact  
id  name  
1  Alice  
2  Bob  
3  Camille  
4  Dov  

Table: companyContactMapping  
id  companyID  contactID  
1  1  3  
2  3  4  
3  2  1  
4  3  2  
5  3  3  

Note that the mapping table comprises only key values: its own primary key, plus two foreign key columns—one each for rows in the company table and rows in the contact table. With the addition of the mapping table, the tables are in 1NF. With a bit of studying (via queries that are easily coded into SQL), it becomes clear that Alice works at BoBoCo, Bob and Dov work at CapCorp, and busy Camille holds down jobs at both AlphaCo and CapCorp. The mapping table makes possible a many-to-many relationship.

If the relationship is to be one-to-many instead of many-to-many (if we assume that while a company can have several employees, a person can only work for one company), the table structure is even simpler. We can maintain 1NF by putting a column in the contact table that establishes a link to the company table, like this:

Table: company
Camille no longer has a place at CapCorp, but we have one table fewer and our database still exhibits 1NF.

6.3.2 Further Normal Forms

There is more to database normalization than 1NF, but the higher normal forms become increasingly academic and hard to conceptualize. Practically speaking, if you obey the following rules of 1NF, to wit, you will have an efficient database in all but a few obscure cases.

- Your tables' contents are as atomic as they can be while retaining their usefulness.
- There are no repeating groups in your tables.

For information on the higher normal forms, consult *Database Design for Mere Mortals*, by Michael Hernandez.

6.4 Deciding on a Table Schema

We know that we're going to build a database that serves as a repository of financial data. The database will change over time as checks are written and withdrawals made, and so will be somewhat transaction oriented. It also, though, has a reporting requirement: The user will need to examine the data in the database without making changes to it.

But what, specifically, is in the database? At this point in the design process, we take an initial shot at listing values that will be stored. Here's an initial list for our global bookkeeping application:

- Information about banks,
- Information about accounts,
- Information about transactions (checks, withdrawals, and interaccount transfers), and
- Information about currencies.
In this part of the process, we might think of information such as the net value of all existing accounts and consider storing such information in the database. It is important that such information should not be stored in the database because it's calculated. We should rely on the software elsewhere in our application to come up with such values for us—and indeed we will write a business-logic layer that provides a net value figure.

### 6.4.1 Initial Table Specifications

Once you have that preliminary list written, you can start to think about converting it to more concrete entities of the kind that can be built into the structure of a database. Specifically, we need to convert our list of information categories into specifications for tables, column headers, and datatypes.

A lot of the time, your general information categories will correspond directly to table names. In this application, for example, we certainly will want to have a table that lists the banks with which our client does business. We'll name that table `ACCT_institutions`, on the logic that the database server might work with applications other than the global bookkeeping one and, we should have a standard prefix (`ACCT_`) to distinguish the tables that interest us. Furthermore, it's possible that the customer would want to record accounts at brokerages, mortgage companies, and other financial companies. Hence, `ACCT_institutions` makes sense as a name.

What data defines a bank? The name, of course. Also the street address, city, state, postal code, country, fax number, and phone number. We should record the name of a contact person at the bank, too. These are our column headers (most of them, anyway, more in a moment). And let's refer to them as "institutions," too, so we can store information about mortgage companies and credit card services as well, and refer to it in an intuitive way. Explicitly stated, the elements that define a financial institution are:

```plaintext
institutionName
streetAddress
city
state
postcode
country
contact
```

These will become columns in `ACCT_institution`, the table that stores data about financial institutions.

Every table should have a primary key, which is a special column header for which all rows have a unique, non-null value. The easiest way to establish a primary key is to establish a column called `id`, assign it the integer datatype (technically, it's the `INT` datatype in MySQL), and tell the database server to automatically increment it as rows are added to the table. Don't worry about the details of how to do that right now.

The next step is to decide on datatypes and sizes for all of these column headers. This decision is something of a balancing act, because you never want your table to be unable to
accommodate an important piece of data, but neither do you want to waste storage space with columns that are much larger than they need to be.

A complete treatment of MySQL datatypes is outside the scope of this book, and the details of the datatypes available in the database server you choose to use may be different anyway. Refer to your database server's documentation for further information. You'll find some links to documentation for popular servers at the end of this chapter. Be aware, though, that the varchar datatype is a very flexible datatype that essentially holds a character string of some specified maximum length. A column that is to be given the varchar type with a maximum length of 50 characters is defined as varchar(50).

In this table, the revised list of columns, complete with proposed datatypes, looks like this:

```
id: integer
institutionName: varchar(100)
streetAddress: varchar(100)
city: varchar(30)
state: varchar(20)
postcode: varchar(100)
country: varchar(30)
contact: integer
```

Why is the contact column an integer? It would be simple to declare it as another shortish varchar field, but if you think about the nature of contacts they're people, with their own names, direct phone lines, e-mail addresses, and so on—most of the information that defines them wouldn't really fit into this table. Although we could associate the name Ruth-Anne Krinklemeyer with the institution record itself, that would enable us to ask to speak to only Ruth-Anne when we called the general bank number. It would be far more useful to recognize (and record) Lee-Anne as a separate entity in the database. Specifically, she should be represented by a row in another table, presumably one called ACCT_contacts. This is a relationship between tables, and such relationships are what's special about relational database servers.

The link between a given row in the ACCT_institutions table and the row corresponding to the relevant contact in the ACCT_contacts table is established by an integer. We store an integer in the contact column of the ACCT_institutions table. This integer corresponds to a value in the id column, the primary key, of the ACCT_contacts table. Because the contact column of the ACCT_institutions table is used this way, it's called a foreign key. This is how we establish a relationship, and hence is why we call this sort of database a relational database.

In databases as in life, relationships introduce a whole set of special problems to solve, as well as a way of dealing with other problems. These tradeoffs are addressed in Section 6.3.

Continuing with our specifications for the ACCT_institutions table, it's true that not all countries have states, and not all countries have postal codes. On the other hand, all banks have names, and all banks are located in one country or another. This sort of analysis
is required when deciding which fields should be able to be null, and which must have values. It would seem that the state, postcode, faxNumber, phoneNumber, and contact columns could legitimately contain null values, on the logic that it’s quite likely that the customer wouldn’t know a bank’s fax number or that he wouldn’t know anyone in particular there. No other column could contain null without compromising the validity of the data in the table.

Figuring out the important data in describing a financial institution is simple enough—you just think about what you’d need if you wanted to contact a particular bank, and add each element to a list. On other jobs, the task of naming columns is more daunting. In those cases, it’s often helpful to look at the documents used to describe the entity. Paper checks, purchase orders, engineering change notices, index cards, and lots of other office paperwork can ease the process of identifying key pieces of information.

Better yet, if the customer is replacing an older software system (perhaps even with an existing database), have a look at its design. No doubt business requirements have changed since it was created, but a look at its table structures will help keep you from forgetting about important data.

6.4.2 Further Table Specifications

Our application will require a number of other tables. The details of these tables are discussed in greater detail when we create them using SQL statements, but at this point in the design process it is important to have a solid understanding of the significance of each table in the database, and how each table links to other tables. The best way to do this is with a diagram of the database schema. It’s possible to draw such a diagram by hand, but it’s easier if you use a documentation and design tool such as CASE Studio (Figure 6.2).

6.5 Translating the Schema into SQL

Eventually, you have to make the leap from theoretical design to practical implementation. When building a database, that means writing some SQL statements that actually create the tables you’ve been hypothesizing about and establish the required relationships among them. This section explains how to write SQL statements that do what earlier sections of this chapter have discussed.

These statements were written for MySQL 4—they were tested on version 4.01b, to be exact. They, or close approximations of them, should work on pretty much any SQL-compliant database, including all of those in PEAR DB compatibility list. To provide exact
Figure 6.2: The complete database schema for our accounting application shows relationships among tables.

SQL statements for all popular database servers would take a lot of space, so remember that this is the last chapter in which our application is concerned with the peculiarities of any database server. All subsequent chapters will either refer to the database through the accessor layer (the presentation, business logic, and elsewhere layers) or through the PEAR DB classes (the accessor layer itself).

All of the table-creation statements appear together in the file currawongTables.sql. The easiest way to run the statements quickly is from a command line, like this:

```
mysql < currawongTables.sql
```

Alternately, you can open the file in a visual environment, such as MySQLCC, and execute the statements from there.

The contents of currawongTables.sql are dissected in the following sections.
6.5 Translating the Schema into SQL

6.5.1 Creating the ACCT_account Table

The ACCT_account table contains facts about categories into which income and expenses are grouped, things like payables, receivables, office rent, and travel expenses. It should not be confused with the ACCT_bank_accounts table, which holds data about accounts at financial institutions that represent assets or debts. Here's the code that creates ACCT_account table, interspersed with comments:

```
DROP TABLE IF EXISTS ACCT_account;
```

The DROP TABLE IF EXISTS statement gets rid of the table if a previous version of it exists. If no such previous version exists, the statement does nothing. DROP statements like this are handy if you expect to be creating your collection of tables several times over, with different characteristics each time—and you probably will, in the course of a typical development job. DROP statements are good practice. On the other hand, they are a kind of delete statement, so you might want to keep backups of your tables and their contents if you're not 100 percent sure of what you're doing.

```
CREATE TABLE 'ACCT_account'
('id' int NOT NULL AUTO_INCREMENT ,
 'name' varchar(100) NOT NULL DEFAULT '' ,
 'parent' int DEFAULT '' ,
 PRIMARY KEY ('id'))
TYPE=InnoDB;
```

The CREATE TABLE clause sets up a table with an automatically incremented id column, which is later designated the primary key. There are also a couple of other data columns. By setting the table type to InnoDB, we make it possible for other tables, the ones containing the transaction register, for example, to use the id column of this table as a foreign key. A foreign key in an InnoDB table must point to another InnoDB table.

6.5.2 Creating the ACCT_trans_type Table

The ACCT_trans_type table holds information about transaction types, such as cash deposits, checks, electronic transfers, and credit card charges. This table, once populated with standard values, shouldn't change much, though it might get a new row from time to time, particularly as new kinds of accounts are opened.

Here is the code for creating the ACCT_trans_type table:

```
DROP TABLE IF EXISTS ACCT_trans_type;
```

```
CREATE TABLE 'ACCT_trans_type'
('id' int NOT NULL AUTO_INCREMENT ,
 'name' varchar(25) NOT NULL DEFAULT '' ,
 PRIMARY KEY ('id')) TYPE=InnoDB;
```
There's nothing complicated to this drop-and-create sequence. The two statements simply get rid of the table if it exists already, then create it afresh as an InnoDB table containing only an auto-incremented primary key column and a text name.

### 6.5.3 Creating the ACCT_acct-type Table

The ACCT_acct-type table contains a list of categories into which we group financial accounts. This table is meant to contain rows designating varieties of account, such as savings, checking, and mortgage.

Here's the creation code:

```sql
DROP TABLE IF EXISTS ACCT_acct_type;

CREATE TABLE 'ACCT_acct_type'
('id' int NOT NULL AUTO_INCREMENT ,
 'name' varchar(15) NOT NULL DEFAULT " ,
 PRIMARY KEY ('id')) TYPE=InnoDB;
```

It's straightforward, simply dropping any extant table of the required name and replacing it with a simple two-column table. The ACCT_acct-type table is of type InnoDB so other tables can use its id column as their foreign keys and expect to have relationships properly enforced.

### 6.5.4 Creating the ACCT_currency Table

The ACCT_currency table contains the currencies in which transactions take place, as well as their values (exchange rates) relative to the U.S. dollar. Here is the code for setting up the table:

```sql
DROP TABLE IF EXISTS ACCT_currency;

CREATE TABLE 'ACCT_currency'
('id' int NOT NULL AUTO_INCREMENT ,
 'country' varchar(100) NOT NULL DEFAULT " ,
 'name' varchar(100) NOT NULL DEFAULT " ,
 'abbreviation' varchar(100) NOT NULL DEFAULT " ,
 'xRate' real NOT NULL DEFAULT " ,
 'updated' timestamp ,
 PRIMARY KEY ('id'))
TYPE=InnoDB;
```

The interesting bit in this creation statement is the updated column, which is of datatype timestamp. The timestamp datatype contains, logically enough, information about a date and time, but the great thing about it is that columns of type timestamp are automatically updated to the current date/time value whenever the row containing them is updated.
In this case, the updated column will contain a piece of data that indicates when any column in the row was last changed. Typically, this will correspond to the date and time the exchange rate (contained in the xRate column) was last modified.

6.5.5 Creating the ACCT_payee Table

The ACCT_payee table holds information about people and companies with whom or with which transactions are concluded. Here is how it's set up:

```sql
DROP TABLE IF EXISTS ACCT_payee;

CREATE TABLE 'ACCT_payee'
('id' int NOT NULL AUTO_INCREMENT ,
 'name' varchar(100) NOT NULL DEFAULT "",
 'streetAddress' varchar(100) DEFAULT "",
 'city' varchar(100) DEFAULT "",
 'state' varchar(100) DEFAULT "",
 'postcode' varchar(100) DEFAULT "",
 'country' varchar(100) DEFAULT "",
 'contact' int DEFAULT "",
 PRIMARY KEY ('id')
) TYPE=InnoDB;
```

The table is quite unremarkable, other than that it's an InnoDB table so that other tables may use its id column as a foreign key.

6.5.6 Creating the ACCT_institution Table

The ACCT_institution table holds contact information for banks, lenders, brokers, and other financial institutions. It's basically another name-and-address table of the type that has appeared a couple of times in this chapter already. Here's the creation code:

```sql
DROP TABLE IF EXISTS ACCT_institution;

CREATE TABLE 'ACCT_institution'
('id' int NOT NULL AUTO_INCREMENT ,
 'institutionName' varchar(100) NOT NULL DEFAULT "",
 'streetAddress' varchar(100) NOT NULL DEFAULT "",
 'city' varchar(30) NOT NULL DEFAULT "",
 'state' varchar(20) DEFAULT "",
 'postcode' varchar(100) DEFAULT "",
 'country' varchar(30) NOT NULL DEFAULT "",
 'contact' int DEFAULT "",
 PRIMARY KEY ('id')
) TYPE=InnoDB;
```
6.5.7 Creating the ACCT_bank_account Table

The ACCT_bank_account table contains information about accounts at institutions, as distinct from the "accounting accounts" contained in the ACCT_accounts table.

The ACCT_bank_account table is interesting in that it is the first table in this database with foreign keys. A bank account, of course, is associated with a financial institution, a currency, and an account type. Those relationships are reflected in the table definition statement shown here:

```
DROP TABLE IF EXISTS ACCT_bank_account;

CREATE TABLE 'ACCT_bank_account'
('id' int NOT NULL AUTO_INCREMENT,
 'institution' int NOT NULL DEFAULT '',
 'number' varchar(100) NOT NULL DEFAULT '',
 'description' varchar(100) NOT NULL DEFAULT '',
 'currency' int NOT NULL DEFAULT '',
 'type' int NOT NULL DEFAULT '',
 PRIMARY KEY ('id'),
 INDEX (institution),
 FOREIGN KEY (institution)
 REFERENCES ACCT_institution(id)
 ON UPDATE CASCADE ON DELETE RESTRICT,
 INDEX (currency),
 FOREIGN KEY (currency)
 REFERENCES ACCT_currency(id)
 ON UPDATE CASCADE ON DELETE RESTRICT,
 INDEX (type),
 FOREIGN KEY (type)
 REFERENCES ACCT_acct_type(id)
 ON UPDATE CASCADE ON DELETE RESTRICT)
TYPE=InnoDB;
```

In defining a foreign key, four operations take place. First, we index the column that is to be defined as a foreign key:

```
INDEX (institution),
```

When a column is indexed, it is ordered for fast searching. Then, we declare explicitly that it is a foreign key:

```
FOREIGN KEY (institution)
```

Then, we establish a connection to a column in a remote table (which, because this is an InnoDB table, must also be of InnoDB type):

```
REFERENCES ACCT_institution(id)
```

Note that the id column of the ACCT_institution table is a primary key, so there can be no ambiguity about which ACCT_institution row is identified by a given value in the
ACCT_bank_account table. Finally, the rules governing the relationship between the two tables are defined:

ON UPDATE CASCADE ON DELETE RESTRICT

That's actually two distinct rules. The first, ON UPDATE CASCADE, indicates that when the value of the referenced field changes, the value of the referring field should change in the same way. That is, if the id value of the referenced row in the ACCT_currency table changes, the id value of the referring row in the ACCT_bank_account table should change to the same value. By including ON UPDATE CASCADE, we ensure that the reference remains valid even if the referenced table changes.

The second rule, ON DELETE RESTRICT, describes what should happen if someone or something tries to delete the referred-to row. The RESTRICT constraint prevents the deletion from going ahead. To put it another way, an attempt to delete a row that is referred to from a table set up with ON DELETE RESTRICT will encounter an error message.

These kinds of restrictions are new with the InnoDB table type. Other table types, such as the very popular MyISAM type, do not enforce foreign key restrictions.

6.5.8 Creating the ACCT_register Table

The ACCT_register table contains information about transactions that take place among the accounts, such as deposits, checks, transfers, interest payments, and so on. Because it's concerned with describing activities, rather than entities, it stands to reason that ACCT_register would be full of foreign keys. It is:

DROP TABLE IF EXISTS ACCT_register;

CREATE TABLE 'ACCT_register'  
('id' int NOT NULL AUTO_INCREMENT , 
'bank_account' int NOT NULL DEFAULT '', 
'number' int DEFAULT '', 
'type' int NOT NULL DEFAULT '', 
'payee' int NOT NULL DEFAULT '', 
'account' int NOT NULL DEFAULT '', 
'date' datetime NOT NULL DEFAULT '', 
'memo' varchar(100) DEFAULT '', 
'amount' real NOT NULL DEFAULT '', 
'reconciled' bool NOT NULL DEFAULT '', 
PRIMARY KEY ('id'), 
INDEX (bank_account), 
FOREIGN KEY (bank_account) 
REFERENCES ACCT_bank_account(id) 
ON UPDATE CASCADE ON DELETE RESTRICT, 
INDEX (type),
Four separate foreign keys are established in this creation statement. Links are made to the ACCT_bank_account, ACCT_trans_type, ACCT_payee, and ACCT_account tables, all of which are central to defining the nature of a financial transaction. In all cases, though, it's the same four-step process to establish a foreign key: INDEX, then FOREIGN KEY, then REFERENCES, then ON UPDATE.
all other company details in the company table, we'd violate the principle of putting everything into the database exactly once. We would have created a maintenance problem, for two reasons:

1. If the company ever changes its name, we’d have to make adjustments not only to the company table, but also potentially to many rows in the contacts table.

2. There would be a risk of the company name value not matching in the two tables, or even in various rows in the contacts table. One contact could be recorded as an employee of “Bergman International;” another might have “Bergman Inc” in the company field. Though logically equivalent to a human being, these two values are not the same when it's time to write a query.

So, we must rely on a different method of establishing relationships between records in the contact table and records in the company table. This is what keys are for.

Recall that when we create a table, we endow it with a special column that's usually defined as an auto-incrementing integer. In populating the table, we never fill this column explicitly, and instead rely on the database server software to populate it automatically as we add columns.

To make a reference between a row in one table and a row in another, the referring row contains a column for a foreign key. A foreign key is a value, stored in one table, that equates to a primary key in another table. In the case of the relationship between contacts and companies, the two tables would look like this:

Table contact
- id
- fname
- lname
- address
- phone
- companyID

Table company
- id
- companyName
- address
- phone

In each table, the id column is designated the primary key.

The companyID column in the contacts table contains a value that appears in the id column of the company table. The companyID column is therefore a foreign key. This relationship is established at table-creation time by means of a FOREIGN KEY... REFERENCES sequence. Here are SQL statements that create the two tables:

```
CREATE TABLE 'company'
('id' int NOT NULL AUTO_INCREMENT ,
 'companyName' varchar(30) NOT NULL DEFAULT '',
 'address' varchar(50) NOT NULL DEFAULT '',
 'phone' varchar(20) NOT NULL DEFAULT '',
 PRIMARY KEY ('id'))
TYPE=InnoDB
```
Chapter 6: The Persistence Layer

6.6 Populating the Tables

This book's supporting files include one called populator.sql, which fills the new tables with standard values that will enable you to use Currawong Accounting right away.

The easiest way to run the populator script is from a command line, like this:

```
mysql < populator.sql
```

There's no point in listing the contents of populator.sql here, because they're very repetitive. The file comprises about a hundred INSERT statements, like this:

```
INSERT INTO ACCT_currency (country, name, abbreviation, xrate) values('Hong Kong', 'Dollar', 'HKD', 0.13);
```

That statement inserts the specified values into the specified columns of ACCT_currency.

Sequence matters. "Hong Kong" goes into the country column because country is first in the columns list, and "Hong Kong" is first in the countries list.

The sequence in which tables are populated by the statements in populator.sql also is important. Have a look at the file and notice that ACCT_bank_account and ACCT_register are among the tables populated last. That's because they depend on making reference to the contents of other tables, such as ACCT_currency. Without an ACCT_currency primary key value to refer to, you could not add a row to ACCT_register.
6.7 Questions and Exercises

1. Briefly state what's required for each of the three most important normal forms (1NF through 3NF). Of the requirements, which would you say is most important to efficiency, and why?

2. What might be a good use for a composite primary key (one that comprises two or more columns)?

3. Why do we usually have a dedicated id column serving as the primary key? What, if anything, is wrong with using a real piece of data?

4. What are the implications of choosing an open-source or GNU Public License (GPL) database server to act as the persistence layer in your application? How does the significance of such a decision change in an academic environment versus a commercial one?