9

Database Applications

We have spent the entire book so far discussing the database as if it exists in some sort of vacuum. It serves its purpose only when being used by other applications. We should therefore take a look at how the database relates to the other elements of a database application before exploring the details of database application development in various languages. This detour examines conceptual issues important not only to programming with MySQL, but also to programming with any relational database engine. Our look at database programming covers such complex issues as understanding the basic architectures common to Web-oriented database applications and how to map complex programming models into a relational database.

Architecture

Architecture describes how the different components of a complex application relate to one another. A simple Web application using Perl to generate dynamic content has the architecture shown in Figure 9-1. This architecture describes four components: the Web browser, the Web server, the Perl CGI engine, and the MySQL database.

FIGURE9-1.BMP

Figure 9-1. . The architecture of a simple Web application

Architecture is the starting point for the design of any application. It helps you identify at a high level all of the relevant technologies and what standards those technologies will use to integrate. The architecture in Figure 9-1, for example, shows the Web browser talking to the server using HTTP.

As we will cover in the later chapters of this section, MySQL exposes itself through a variety of APIs tailored to specific programming languages. Java applications access MySQL through JDBC; Python applications through the Python DB-API, etc. The architecture above clearly shows to any observer that the application in question will use the Perl DBI API to access MySQL.

There are numerous architectures used in database applications. In this chapter, we will cover the three most common architectures: client/server, distributed, and Web. Though one could argue that they are all variations on a theme, they do represent three very different philosophical approaches to building database applications.

Client/Server Architecture

At its simplest, the client/server architecture is about dividing up application processing into two or more logically distinct pieces. The database makes up half of the client/server architecture. The database is the 'server'; any application that uses that data is a 'client.' In many cases, the client and server reside on separate machines; in most cases, the client application is some sort of user-friendly interface to the database. Figure 9-2 provides a graphical representation of a simple client/server system.

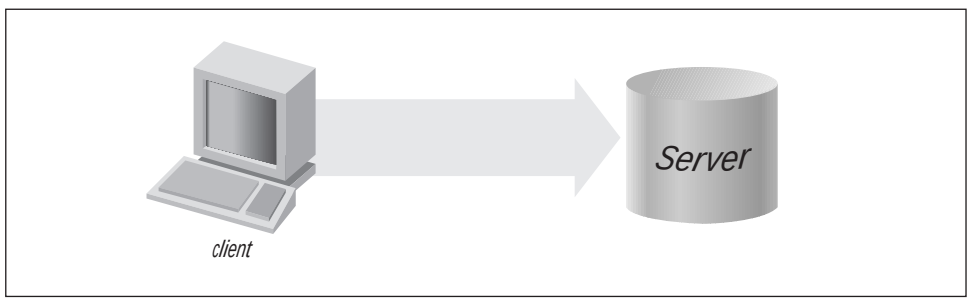

Figure 9-2. The client/server architecture

You have probably seen this sort of architecture all over the Internet. The Web, for example, is a giant client/server application in which the Web browser is the client and the Web server is the server. In this scenario, the server is not a relational database server, but instead a specialized file server. The essential quality of a server is that it serves data in some format to a client.

Application Logic

Because client/server specifically calls out components for user interface and data processing, actual application processing is left up to the programmer to integrate. In other words, client/server does not provide an obvious place for a banking application to do interest calculations. Some client/server applications place this

kind of processing in the database in the form of stored procedures; others put it in the client with the user interface controls. In general, there is no right answer to this question.

Under MySQL, the right answer currently is to put the processing in the client due to the lack of stored procedure support in MySQL. Stored procedures are on the MySQL to-do list, and—perhaps even by the time you read this book—stored procedures will eventually be a viable place for application logic in a client/server configuration. Whether or not MySQL has stored procedures, however, MySQL is rarely used in a client/server environment. It is instead much more likely to be used with the Web architecture we will describe later in this chapter.

Fat and Thin Clients

It used to be that there were two kinds of clients: fat clients and thin clients. A fat client was a client in a client/server applications that included application processing; a thin client was one that simply had user interface logic. With the advent of Web applications, we now have the term ultra-thin to add to the list. An ultra-thin client is any client that has only display logic. Controller logic—what happens when you press "Submit"—happens elsewhere. In short, an ultra-thin client is a Web form.

The advantage of an ultra-thin client is that it makes real the concept of a ubiquitous client. As long as you can describe the application layout to a client using some sort of markup language, the client can paint the UI for a user without the programmer needing to know the details of the underlying platform. When the UI needs to respond to a user action, it sends information about the action to another component in the architecture to respond to the action. Client/server, of course, has no such component.

Distributed Application Architecture

The distributed application architecture provides a logical place where application logic is supposed to occur, but it does not provide a place for UI controller logic. Figure 9-3 shows the layout of an application under the distributed application architecture.

FIGURE9-3.BMP

Figure 9-3. . The distributed application architecture

As you can see, this architecture is basically the client/server architecture with a special place for application logic—the middle tier. This small difference, however, represents a major philosophical shift from the client/server design. It says, in short, that it is important to separate application logic from other kinds of logic.

In fact, placing application logic in the database or in the user interface is a good way to hinder your application's ability to grow with changing demands. If, for example, you need a simple change to your data model, you will have to make significant changes to your stored procedures if your application logic is in the database. A change to application logic in the UI, on the other hand, forces you to touch the UI code as well and thus risk adding bugs to systems that have nothing to do with the changes you are introducing.

The distributed application architecture thus does two things. First, it provides a home for application processing so that it does not get mixed in with database or user interface code. The second thing it does, however, is make the user interface independent of the underlying data model. Under this architecture, changes to the data model affect only how the middle tier gets data from and puts it into the database. The client has no knowledge of this logic and thus does not care about such changes.

The distributed application architecture introduces two truly critical elements. First of all, the application logic tier enables the reuse of application logic by multiple clients. Specifically, by calling out the application logic with well-defined integration points, it is possible to reuse that logic with user interfaces not conceived when the application logic was written.

The second, not so obvious thing this architecture brings to applications is the ability to provide easy support for fail-over and scalability. The components in this architecture are logical components, meaning that they can be spread out across multiple actual instances. When a database or an application server introduces clustering, it can act and behave as a single tier while spreading processing across multiple physical machines. If one of those machines goes down, the middle-tier itself is still up and running.

Complex transactions are a hallmark of distributed applications. For that reason, MySQL today makes a poor backend for this architecture. As support for transactions in MySQL matures, this state of affairs may change.

Web Architecture

The Web architecture is another step in evolution that appears only slightly different than the distributed application architecture. It makes a true ultra-thin client possible by providing only display information in the form of HTML to a client. All controller logic occurs in a new component, the Web server. Figure 9-4 illustrates the Web architecture.

FIGURE9-4.BMP

Figure 9-4. . The Web application architecture

The controller comes in many different forms, depending on what technologies you are using. PHP, CGI, JSP, ASP, ColdFusion, and WebObjects are all examples of technologies for processing user events. Some of these technologies even break things up further into content creation and controller logic. Using a content management system like OpenMarket, for example, your JSP is nothing more than a tool for dynamically building your HTML. The actual controller logic is passed off to a servlet action handler that performs any application server interaction.

The focus of this book will be the Web architecture since it is the most common architecture in which MySQL is used. We will use both the vision of the Web architecture shown in Figure 9-4 and a simpler one in which the application logic is embedded with controller logic in the Web server. The simpler architecture is mostly relevant to MySQL applications since MySQL performs best for heavy read applications—applications without complex application logic.

Connections and Transactions

Whatever architecture you are using, the focus of this book lies at the point where your application talks to the database. As a database programmer, you need to worry about how you get data from and send it to your database. As we mentioned earlier, the tool to do that is generally some sort of database API. Any API, however, requires a basic understanding of managing a connection, the transactions under that connection, and the processing of the data associated with those transactions.

Connections

The starting point of your database interaction is in making the connection. The details behind what exactly it is to be a connection vary from API to API. Nevertheless, making a connection is basically establishing some sort of link between your code and the database. The variance comes in the form of logical and physical connections. Under some APIs, a connection is a physical connection—a network link is established. Other APIs, however, may not establish a physical link until long after you make a connection, to ensure that no network traffic takes place until you actual need the connection.

The details about whether or not a connection is logical or physical generally should not concern a database programmer. The important thing is that once a connection is established, you can use that connection to interact with the database.

Once you are done with your connection, you need to close it and free up any resources it may have used. It stands to reason that before you actually issue a

query, you should first connect to the database. It is not uncommon, however, for people to forget the other piece of the puzzle—cleaning up after themselves. You should always free up any database resources you grab the minute you are done with them. In a long-running application like an Internet daemon process, a badly written system can eat up database resources until it locks up the system.

Part of cleaning up after yourself involves proper error handling. Better programming languages make it harder for you to fail to handle exceptional conditions (network failure, duplicate keys on insert, SQL syntax errors, etc.); but, regardless of your language of choice, you must make sure that you know what error conditions can arise from a given API call and act appropriately for each exceptional situation.

Transactions

You talk to the database in the form of transactions.* A simple description of a database transaction is one or more database statements that must be executed together, or not at all.A bank account transfer is a very good example of a complex transaction. In short, an account transfer is actually two separate events: a debit of one account and a credit to another. Should the database crash after the debit occurs but before the credit, the application should be able to back out of the debit. A database transaction enables a programmer to mark when a transaction begins, when it ends, and what should happen should one of the pieces of the transaction fail.

Until recently, MySQL had no support for transactions. In other words, when you executed a SQL statement under old versions of MySQL, it took effect immediately. This behavior is still the default for MySQL. Newer versions of MySQL, however, support the ability to use transactions with certain tables in the database. Specifically, the table must use a transaction-safe table format. Currently, MySQL supports two transaction-safe table types: BDB (Berkeley DB) and InnoDb.

In Chapter 4, we described the MySQL syntax for managing transactions from the MySQL client command line. Managing transactions from within applications is often very different. In general, each API will provide a mechanism for beginning, committing, and rolling back transactions. If it does not, then you likely can follow the command line SQL syntax to get the desired effect.

* Even if you are using a version of MySQL without support for transactions, each statement you send to the database can, in a sense, be thought of as an individual transaction. You simple have no option to abort or package multiple statements together in a complex transaction.

Transaction Isolation Levels

Managing transactions may seem simple, but there are many issues you need to consider when using transactions in a multi-user environment. First of all, transactions come with a heavy price in terms of performance. MySQL did not originally support transactions because MySQL's goal was to provide a fast database engine. Transactions seriously impact database performance. In order to understand how this works, you need to have a basic understanding of transaction isolation level.

A transaction isolation level basically determines what other people see when you are in the middle of a transaction. In order to understand transaction isolation levels, however, you first need to understand a few common terms:

dirty read
: A dirty read occurs when one transaction views the uncommitted changes of another transaction. If the original transaction rolls back its changes, the one that read the data is said to have "dirty" data.

repeatable read
: A repeatable read occurs when one transaction always reads the same data from the same query no matter how many times the query is made or how many changes other transactions make to the rows read by the first transaction. In other words, a transaction that mandates repeatable reads will not see the committed changes made by another transaction. An application needs to start a new transaction to see those changes.

phantom read
: A phantom read deals with changes occurring in other transactions that would result in the new rows matching your transaction's `WHERE` clause. Consider, for example, a situation in which you have a transaction that reads all accounts with a balance of less than $100. Your transaction performs two reads of that data. Between the two reads, another transaction adds a new account to the database with no balance. That account will now match your query. If your transaction isolation allows phantom reads, you will see the new "phantom" row. If it disallows phantom reads, then you will see the same set of rows each time.

MySQL supports the following transaction isolations levels:

READ UNCOMMITTED
: The transaction allows dirty reads, non-repeatable reads, and phantom reads.

READ COMMITTED
: The transaction disallows dirty reads, but it allows non-repeatable reads and phantom reads.

REPEATABLE READ
: Committed, repeatable reads as well as phantom reads are allowed. Non-repeatable reads are not allowed.

SERIALIZABLE
: Only committed, repeatable reads are allowed. Phantom reads are specifically disallowed.

As you climb the transaction isolation chain, from no transactions to serializable transactions, you decrease the performance of your application. You therefore need to balance your data integrity needs with your performance needs. In general, READ COMMITTED is as high as an application wants to go, except in a few very exceptional cases.

Using READ UNCOMMITTED

One mechanism of getting the performance of READ UNCOMMITTED but the data integrity of READ COMMITTED is to make a row's primary key the normal primary key plus a timestamp reflecting the time in milliseconds when the row was last updated. When an application performs an update on the underlying row in the database, it updates that timestamp but uses the old one in the WHERE clause:

```
UPDATE ACCOUNT
SET BALANCE = 5.00, LAST_UPDATE_TIME = 996432238000
WHERE ACCOUNT_ID = 5 AND LAST_UPDATE_TIME = 996432191119
```

If this transaction has dirty data, the update will fail and throw an error. The application can then re-query the database for the new data.

Object/Relational Modeling

Accessing a relational database from an object-oriented environment exposes a special paradox: the relational world is entirely about the manipulation of data while the object world is about the encapsulation of data behind a set of behaviors. In an object-oriented application, the database serves as a tool for saving objects across application instances. Instead of seeing the query data as a rowset, an object-oriented application sees the data from a query as a collection of objects.

The most basic question facing the object-oriented developer using a relational database is how to map relational data into objects. Your immediate thought might be to simply map object attributes to fields in a table. Unfortunately, this approach does not create the perfect mapping for several reasons.

- Objects do not store only simple data in their attributes. They may store collections or relationships with other objects.

- Most relational databases—including MySQL—have no way of modeling inheritance.

Rules of Thumb for Object/Relational Modeling

- Each persistent class has a corresponding database table.
- Object fields with primitive datatypes (integers, characters, strings, etc.) map to columns in the associated database table.
- Each row from a database table corresponds to an instance of its associated persistent class.
- Each many-to-many object relationship requires a join table just as database entities with many-to-many relationships require join tables.
- Inheritance is modeled through a one-to-one relationship between the two tables corresponding to the class and subclass.

Think about an address book application. You would probably have something like the `address` and `person` tables shown in Figure 9-5.

The least apparent issue facing programmers is one of mindset. The basic task of object-oriented access to relational data is to grab that data and *immediately* instantiate objects. An application should only manipulate data through the objects. Most traditional programming methods, including most C, PowerBuilder, and VisualBasic development, require the developer to pull the data from the database and then process that data. The key distinction is that in object-oriented database programming, you are dealing with objects, not data.

Figure 9-6 shows the object model that maps to the data model from Figure 9-5. Each row from the database turns into a program object. Your application therefore takes a result set and, for each row returned, instantiates a new `Address` or `Person` instance. The hardest thing to deal with here is the issue mentioned earlier: how do you capture the relationship between a person and her address in the database application? The `Person` object, of course, carries a reference to that person's `Address` object. But you cannot save the `Address` object within the `person` table of a relational database. As the data model suggests, you store object relationships through foreign keys. In this case, we carry the `address_id` in the `person table`.

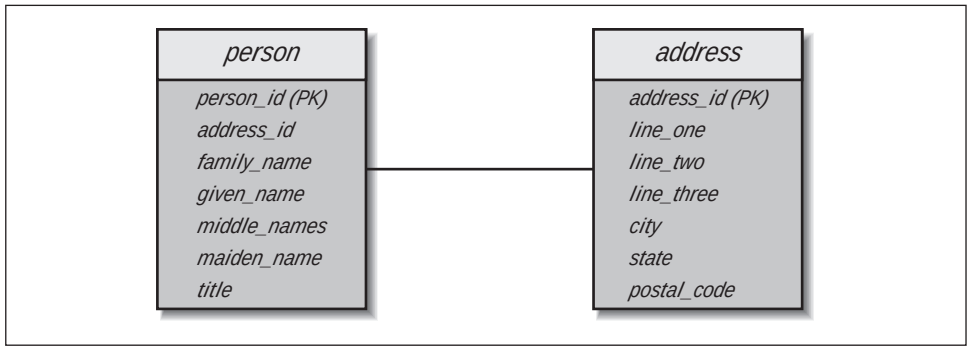

Figure 9-5. The data model for a simple address book application

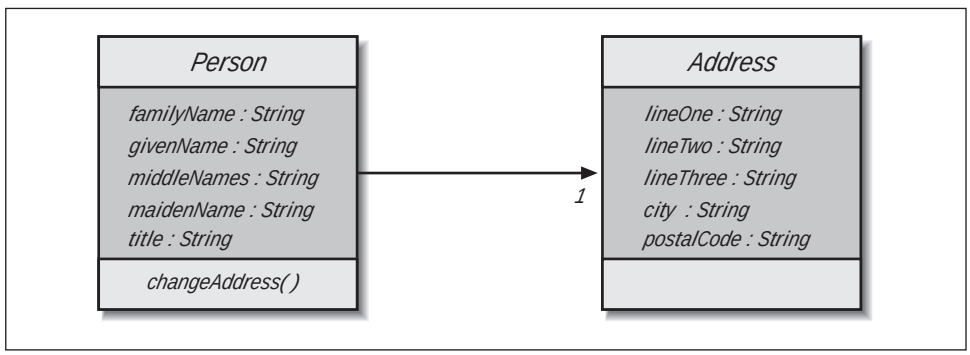

Figure 9-6. The object model supporting a simple address book application

With just a tiny amount of extra complexity to the object model, we can add a world of complexity to the challenge of mapping our objects to a data model. The extra bit of complexity could be to have `Person` inherit from `Entity` with a `Company` class also inheriting from `Entity`. How do we capture an `Entity` separate from a `Person` or a `Company`? The rule we outlined above is actually more of a guideline. In some instances, the base class may be purely abstract and subsequently have no data associated with it in the database. In that instance, you would not have an entity in the database for that class.