**Using Complex Database Queries**

The key to any dynamic Web application is the database and retrieving information from it. In a simple database layout, data is usually stored in a single table. There's no need to query multiple tables looking for data.

In a simple database environment, you see SQL query statements that look like this:

SELECT recipeid, title, poster, shortdesc FROM recipes ORDER BY recipeid DESC

This simple SQL statement just retrieves four data fields from a single table. In more complex applications, data is often split between multiple tables, as you see in our products and categories tables. If we list products from the products table, all we see is a catid value, not the real category name. To extract meaningful information, you need to look up the catid in the categories table to find the full category name. This makes displaying the product information more meaningful, as it just uses words instead of database codes.

Fortunately, SQL allows us to extract related information from two tables within a single SQL query. Because we're using two tables at the same time, the query itself gets a lot more complicated than what you may be used to seeing. But the results are well worth the extra effort.

The list of data fields is more complicated because you must identify which table each data field is from. You can do that by using this format: table.datafield

When you use the FROM clause, you need to list all of the tables used in the query. And the WHERE clause must provide a statement that specifies which data fields should be related in the query.

Let's take a look at an example of this. Here's a query that lists the names and addresses of the customers associated with each order:

SELECT products.description, categories.name, products.quantity FROM products, categories WHERE products.catid = categories.catid;

This example extracts the category name from the categories table and the product description and quantity in stock from the products table to produce a report that actually makes sense. The relation between these two tables is the catid data field.

You need the WHERE clause to let MySQL know to combine the two tables by matching the catid data values between them. Let's test drive this SQL statement and see what it gets us. First, you'll need to insert some test data into the tables:

1. Start the MySQL Console from the WampServer icon.
2. Connect to the store database.

connect store;

1. Insert a few records into the categories table:
2. INSERT into categories (name) VALUES ("produce");
3. INSERT into categories (name) VALUES ("fruit");

INSERT INTO categories (name) VALUES ("bread");

1. Now, insert a few products into the products table:

INSERT INTO products (catid, description, price, quantity) VALUES (1, "eggplant", 1.25, 20);

INSERT INTO products (catid, description, price, quantity) VALUES (1, "tomatoes", 2.00, 15);

INSERT INTO products (catid, description, price, quantity) VALUES (2, "bananas", 1.25, 5);

1. Use the SELECT SQL statements to ensure your data is in the tables.

Now that you have some data, let's test our fancy SQL query. From the MySQL prompt, enter the SQL statement:

SELECT products.description, categories.name, products.quantity FROM products, categories WHERE products.catid = categories.catid;

MySQL processes the query and displays the result set as shown below:

The result of the complex query

These results are much better than just throwing a category code at our customers!

Notice that all the WHERE clause does is match the two related data fields in the two tables. That's a lot of typing to do just to match two data fields that already have the same name. You'd think that MySQL could do that for us. Guess what? SQL uses a special SELECT syntax that does this automatically!

**Joining Tables**

You use the *JOIN* keyword to join two tables together on a common data field name. Here's an example of the same query using the JOIN keyword:

SELECT products.description, categories.name, products.quantity FROM products JOIN categories USING (catid)

The JOIN keyword is in the middle of the two listed tables in the FROM clause. It also requires the USING clause to specify the data field that relates the two tables. You must specify the data field within the parentheses.

**Creating Database Views**

Now you've seen a complex query that can extract data from two tables based on a common data field name. This process also works for multiple tables that share data field names. However, it can get pretty complicated when you're working with multiple tables.

The MySQL InnoDB database engine provides yet another advanced feature to help us out with commonly used complex queries. A database *view* is a snapshot of the database query results presented as a single logical table. The database handles the view as a normal table. You can use the view in SQL queries just as you would a regular table. The data fields in the view consist of data fields from the query result set.

It's important to remember that the view isn't a real table. It's a temporary table that MySQL creates on the fly from data contained in other tables as you query the view.

What's also great about views is they can also include SQL functions and calculations on data, providing a wealth of possibilities. Let's take a look at a sample view:

CREATE VIEW prod\_view AS SELECT products.description, categories.name, products.price, products.quantity, products.price \* products.quantity as value FROM products JOIN categories USING (catid);

This view creates a new view called *prod\_view* from a standard SQL SELECT statement. It includes five data fields: the product description, the name of the product's category, the product's price, the product's quantity in stock, and a calculated data field called *value*. The value field is a result of multiplying the product price and the quantity in stock. The AS keyword allows you to assign a specific name to a data field in the result set. This comes in handy when you have calculated fields in a view.

Okay, let's create this view and see what happens:

1. Start the MySQL Console from the WampServer icon.
2. Connect to the store database.
3. Create the view using the SQL code shown above.

Now let's test drive the new view. From the MySQL Console prompt, enter a simple SQL query to extract the data from the view.

SELECT \* from prod\_view;

If all goes well, you should get a listing of the products along with the newly calculated values.

Creating and testing the prod\_view view

You can now create advanced SQL queries, store them as views, and recall them at any time in your PHP code as simple SQL queries. That's pretty powerful stuff! Okay, enough for today. Let's move on to the Summary and wrap things up.