**Database Transactions**

Returning to our checkout saga, we last left our customer standing at our virtual checkout counter. The customer had just registered (or logged in using an existing account) and confirmed the items in the order. We're now ready to create the final customer order and complete the checkout process.

Creating the final order requires performing three database tasks:

1. Creating a new record in the *orders* table to represent the new customer order.
2. Adding a record to the *order\_items* table for each product in the customer's shopping cart, relating each ordered item to the new order number we created in the orders table.
3. Subtracting the quantity of each product ordered from the quantity data field in the products table for that product to update our inventory.

As you work through the details required to check out your customer, you'll find yourself in somewhat of a programming dilemma. All three of these database functions need to happen, or else you have a database problem on your hands. For example:

* If you create a new record in the orders table without any related records in the order\_items table, you won't have a clue what items the customer purchased for the order.
* If you add items to the order\_items table without subtracting the quantities from the products table, your inventory total will be off, causing problems for future orders.

Trying to tie these three functions together requires an advanced database technique called *transactions*.

Transactions are a crucial element to any relational database system. In a transaction, the database engine handles a group of two or more SQL statements as a single unit. If all of the SQL statements in a transaction succeed, the transaction unit succeeds and everything's fine.

However, if any one of the SQL statements in a transaction fails, you have a problem. Not only should the transaction as a whole fail, but any SQL statements in the transaction that were executed previous to the failed statement need to be undone to return the database back to how it was before you attempted the transaction.

For example, if you create the customer order record in the orders table but then the attempt to subtract the ordered products from the products table fails, you need to remove the order from the orders table and inform the customer of the issue. Having an order entered in the orders table without subtracting the products from the inventory in the products table will cause problems for your inventory.

**Creating Transactions in MySQL**

The MySQL database server only recently provided support for transactions (and it's only available using the InnoDB database engine). However, you need to use special SQL statements to enable transaction support.

By default, every time you enter an SQL statement in MySQL (either via the MySQL Console or using the PHP mysql\_query() function), the MySQL database engine performs the statement automatically. This is called *committing* the statement to the database.

When an SQL statement is committed to the database, it permanently enters the results of the statement into the tables, adds inserted data to the tables, removes deleted data from the tables, and changes updated data in the tables. There's no turning back from a committed SQL statement (aside from restoring the database from a previous backup).

When MySQL automatically commits a statement to the database, it's called *autocommit*. MySQL enables this feature by default.

In a transaction, the MySQL server doesn't autocommit statements. You can perform individual statements, but the server doesn't apply the results permanently. You need to tell it to by using the *COMMIT* statement. Instead, statements are applied in a temporary state, apart from the normal database data.

To perform transactions, you must first disable the autocommit feature in MySQL. You do this using the *set* SQL statement:

set autocommit=0

This disables the autocommit feature for the current session and doesn't affect any other sessions connected to the server. Now no statements are permanent until you perform the *COMMIT* statement. At any time before the COMMIT statement, you can issue a *ROLLBACK* statement to undo the effect of any previously entered SQL statement.

You define a transaction as a block of SQL statements. You start the block using the *START TRANSACTION* statement, and you end it using either the COMMIT or ROLLBACK statement.

**Testing Transactions**

Let's test this feature out so you can visually see how it works. Follow these steps to test transactions in MySQL:

1. Start the MySQL Console by clicking the **WampServer** icon in the system tray. Then select **MySQL**, then **MySQL Console**.
2. Enter the password for your MySQL root user account (the default is no password, so just hit the ENTER key).
3. Go to the store database by entering the command:

use store;

1. Turn off the autocommit feature by entering the command:

set autocommit=0;

1. Start a transaction block by entering the command:

START TRANSACTION;

1. Insert data into the categories table:

INSERT INTO categories (name) VALUES ('test1');

1. Insert a few more test categories into the categories table using the INSERT statement.
2. List the items in the categories table to ensure your new items are there.

SELECT \* FROM categories;

1. Now roll back the transaction to prevent the process of the transaction:

ROLLBACK;

1. List the items in the categories table again to see if your new items are still there:

SELECT \* FROM categories;

By rolling back the transaction, you undid any previous SQL statements entered since the START TRANSACTION statement. Notice that before the ROLLBACK statement, when you display the categories table contents, you see your new entries just fine, including the auto\_increment values MySQL assigned to them. All of that was only a temporary copy for you to see during the transaction. None of the data was actually entered into the table. If any other user logged into the server and displayed the categories data during your transaction, they wouldn't see the data from your transaction.

Now, let's actually commit some data from a transaction. Follow these steps:

1. Start the MySQL Console again.
2. Go to the store database using the *use store;* statement.
3. Disable the autocommit feature using the command:

set autocommit=0;

1. Start a new transaction using the statement:

START TRANSACTION;

1. Enter a new category in the categories table:

INSERT INTO categories (name) VALUES ('frozen food');

1. Commit the transaction into the table:

COMMIT;

1. List the contents of the categories table:

SELECT \* FROM categories;

1. Exit the MySQL Console by using the exit command

Once you enter the COMMIT statement, the data you entered in your transaction appears permanently in the table. If you display the categories table, you'll see an oddity with the way MySQL handles transactions.

The new category you added in the transaction has a catid value that's not in sequence with the other values. It has a sequential value that is after the values temporarily assigned in the rolled-back transactions. Unfortunately, MySQL isn't able to rollback auto\_increment values in transactions. Once MySQL assigns an auto\_increment value to a transaction, even if you roll it back, it assigns the next auto\_increment value.

Transactions are a crucial piece of your checkout process. You'll use them in your PHP code for the final checkout program. However, there's one more feature left to discuss before you can start coding. Follow me to Chapter 3, and let's work on some more database theory.