**Updating the Inventory**

Part of the Food Store application allows your store manager to track product inventory in real time, as customers purchase products. This means that you need to have a system that automatically subtracts purchased products from the quantity data field in the products table.

This requires using the UPDATE SQL statement, along with a little math. If this isn't something you've ever done before in the database world, don't worry—there's not much to it.

The SQL language allows you to perform mathematical operations on table data fields. The mathematical operations apply to any record in the table that meets the WHERE clause filter. So be careful when working with operations.

Here's an example of updating the quantity of a product:

UPDATE products SET quantity = quantity - 1 WHERE prodid = 1;

This statement takes the value contained in the quantity data field, subtracts one from it, and stores it back in the quantity data field of the record. It performs this task on all records that match the WHERE clause. So be careful you remember to include the WHERE clause.

This works great for tracking your inventory, but there's one small challenge you'll run into and have to solve.

**Adding Table Constraints**

When a customer purchases an item, you just add the UPDATE statement to the transaction so it subtracts the appropriate quantity of a product from the running inventory. Unfortunately, the way the table is set up, when the inventory gets to zero, the next update will just happily set the quantity to a negative value.

It would be nice if we could prevent customers from placing an order for a product that's out of stock. If you remember the discussion from the storefront PHP code, before allowing a customer to place a product in his or her shopping cart, you make sure there's enough of the product in stock. However, since you don't immediately remove that amount from the products table quantity data field, a situation could occur when other customers check out before the customer. So when that customer checks out the product is no longer in stock.

You can use a MySQL table constraint to control this situation. The table constraint places a restriction on the table that must be followed. You can force the UPDATE statement to fail if you attempt to subtract more of a product than the quantity in inventory. This requires a little SQL trickery.

By default, when you create the quantity data field as an integer data type, MySQL assigns it as a signed integer. A signed integer can contain both positive and negative values. If the quantity value of a product is zero, and you perform an UPDATE statement that removes one from the quantity value, the new quantity value just becomes -1. This isn't a good way to keep your inventory.

Instead, you need a method to prevent transactions from succeeding if they'd force the quantity value to be negative. This requires using two features of MySQL:

* Creating an unsigned integer data type constraint.
* Setting MySQL to strictly enforce table constraints.

For the first piece of the puzzle, you just need to change the attribute of the quantity data type from a signed integer to an *unsigned* integer. The unsigned constraint only allows positive integer values in the data field. This prevents the data field from containing negative values or processing an UPDATE that would result in a negative value.

You can do this either by using the graphical phpMyAdmin tool or using the MySQL Console command line interface. To change the data type attribute on an existing data field using the MySQL Console, you need to use the *ALTER TABLE* SQL statement.

This powerful SQL statement allows you to alter existing configuration information for a table. You redefine an existing data field within the table by specifying the *CHANGE*option in the ALTER TABLE statement:

ALTER TABLE products CHANGE quantity quantity int unsigned not null;

This statement changes the quantity data field to add the unsigned constraint. Notice that the quantity data field name appears twice in the statement. This looks weird, but that's how you need to do it. Let's perform this on the products table in the store database. Just follow these steps:

1. Start the WampServer and log into the MySQL Console
2. At the prompt, type:

**use store;**

1. At the next prompt, type:

**ALTER TABLE products CHANGE quantity quantity int unsigned not null;**

Now the quantity data field is set so it can only contain positive integer values.

**Forcing Table Constraints**

You might now be lulled into a false sense of security. With the quantity data field now set with the unsigned constraint, you might think that MySQL will prevent the creation of a negative value there. Wrong! Here's an example of what I mean.

Creating a 'negative' number in an unsigned data field

If you attempt to subtract more than the quantity value, something odd happens. Instead of producing an error message, MySQL performs the subtraction. The data value produced isn't what you think it should be. Instead of a negative value, a large positive value appears!

The trouble is that the integer value *wraps around*, creating an extremely large positive value instead of the negative value. This is a feature of unsigned values. Instead of showing a negative number, the value goes to the top of the data value range and starts working downward. This wraparound is due to the configuration of bits in the data field and how the MySQL server performs binary arithmetic.

This feature is an issue with MySQL. Originally, MySQL didn't support table constraints. To provide backward compatibility, by default MySQL permits functions that violate table constraints. You need to change that for your inventory algorithm to work.

The *sql\_mode* setting determines how MySQL handles data validation checks. By default, MySQL doesn't perform validation checks on data in SQL statements, so invalid data can be inserted into a table (such as in the case of our unsigned example above).

To prevent this from happening, you must change the sql\_mode of your MySQL session. By setting the sql\_mode to *STRICT\_ALL\_TABLES*, MySQL will enforce strict data validation checks on all data entry constraints.

The SQL statement you need to use is:

set sql\_mode = 'STRICT\_ALL\_TABLES';

When you set this, MySQL validates all data operations, including the UPDATE statement.

Enforcing data validation checks in MySQL

Now, if an UPDATE statement causes the quantity data field to produce a negative value, the UPDATE statement fails, and no operation occurs on the record. This behavior is exactly what we want in our application.

We can add the sql\_mode setting in our transaction statements, forcing strict data validation checks on our UPDATE statements. If an UPDATE statement attempts to create a negative inventory, the UPDATE will fail, causing the entire transaction to fail.

Now that you've seen the theory behind our checkout process, it's time to put that theory into PHP (and SQL) code. Let's continue on to Chapter 4 and start coding!