This information is modified from the following sources

http://www.roseindia.net/mysql/

http://www.tech-faq.com/mysql-tutorials.shtml

http://www.brainbell.com/tutorials/MySQL/Using_Stored_Procedures.htm

Using stored procedures requires knowing how to execute (run) them. Stored procedures are executed far more often than they are written, so we'll start there. And then we'll look at creating and working with stored procedures.

Executing Stored Procedures

MySQL refers to stored procedure execution as calling, and so the MySQL statement to execute a stored procedure is simply `CALL`. `CALL` takes the name of the stored procedure and any parameters that need to be passed to it. Take a look at this example:

• Input

```
CALL productpricing(@pricelow,
                    @pricehigh,
                    @priceaverage);
```

• Analysis

Here a stored procedure named `productpricing` is executed; it calculates and returns the lowest, highest, and average product prices.

Stored procedures might or might not display results, as you will see shortly.

Creating Stored Procedures

As already explained, writing a stored procedure is not trivial. To give you a taste for what is involved, let's look at a simple examplea stored procedure that returns the average product price. Here is the code:

• Input

```
CREATE PROCEDURE productpricing()
BEGIN
   SELECT Avg(prod_price) AS priceaverage
   FROM products;
END;
```

• Analysis

Ignore the first and last lines for a moment; we'll come back to them shortly. The stored procedure is named `productpricing` and is thus defined with the statement `CREATE PROCEDURE productpricing()`. Had the stored procedure accepted parameters, these would have been enumerated between the `(` and `)`. This stored procedure has no parameters, but the trailing `()` is still required. `BEGIN` and `END` statements are used to delimit the stored procedure body, and the body itself is just a simple `SELECT` statement (using the `Avg()` function learned in Tutorial 12, "Summarizing Data").

When MySQL processes this code it creates a new stored procedure named `productpricing`. No data is returned because the code does not call the stored procedure, it simply creates it for future use.

Note

mysql Command-line Client Delimiters If you are using the mysql command-line utility, pay careful attention to this note.

The default MySQL statement delimiter is `;` (as you have seen in all of the MySQL statement used thus far). However, the mysql command-line utility also uses `;` as a delimiter. If the command-line utility were to interpret the `;` characters inside of the stored procedure itself, those would not end up becoming part of the stored procedure, and that would make the SQL in the stored procedure syntactically invalid.

The solution is to temporarily change the command-line utility delimiter, as seen here:

```
DELIMITER //

CREATE PROCEDURE productpricing()
BEGIN
   SELECT Avg(prod_price) AS priceaverage
   FROM products;
END //

DELIMITER ;
```

Here, `DELIMITER //` tells the command-line utility to use `//` as the new end of statement delimiter, and you will notice that the `END` that closes the stored procedure is defined as `END //` instead of the expected `END;`. This way the `;` within the stored procedure body remains intact and is correctly passed to the database engine. And then, to restore things back to how they were initially, the statement closes with a `DELIMITER ;`.

Any character may be used as the delimiter except for `\`.

If you are using the mysql command-line utility, keep this in mind as you work through this tutorial.

So how would you use this stored procedure? Like this:

• Input

```
CALL productpricing();
```

• Output

```
+--------------+
| priceaverage |
+--------------+
|    16.133571 |
+--------------+
```

• Analysis

`CALL productpricing();` executes the just-created stored procedure and displays the returned result. As a stored procedure is actually a type of function, `()` characters are required after the stored procedure name (even when no parameters are being passed).

Dropping Stored Procedures

After they are created, stored procedures remain on the server, ready for use, until dropped. The drop command (similar to the statement seen Tutorial 21, "Creating and Manipulating Tables") removes the stored procedure from the server.

To remove the stored procedure we just created, use the following statement:

• Input

```
DROP PROCEDURE productpricing;
```

• Analysis

This removes the just-created stored procedure. Notice that the trailing `()` is not used; here just the stored procedure name is specified.

Tip

Drop Only If It Exists `DROP PROCEDURE` will throw an error if the named procedure does not actually exist. To delete a procedure if it exists (and not throw an error if it does not), use `DROP PROCEDURE IF EXISTS`.

Working with Parameters

`productpricing` is a really simple stored procedureit simply displays the results of a `SELECT` statement. Typically stored procedures do not display results; rather, they return them into variables that you specify.

New Term

Variable A named location in memory, used for temporary storage of data.

Here is an updated version of `productpricing` (you'll not be able to create the stored procedure again if you did not previously drop it):

• Input

```
CREATE PROCEDURE productpricing(
   OUT pl DECIMAL(8,2),
   OUT ph DECIMAL(8,2),
   OUT pa DECIMAL(8,2)
)
BEGIN
   SELECT Min(prod_price)
   INTO pl
   FROM products;
   SELECT Max(prod_price)
   INTO ph
   FROM products;
   SELECT Avg(prod_price)
   INTO pa
   FROM products;
END;
```

• Analysis

This stored procedure accepts three parameters: `pl` to store the lowest product price, `ph` to store the highest product price, and `pa` to store the average product price (and thus the variable names). Each parameter must have its type specified; here a decimal value is used. The keyword `OUT` is used to specify that this parameter is used to send a value out of the stored procedure (back to the caller). MySQL supports parameters of types `IN` (those passed to stored procedures), `OUT` (those passed from stored procedures, as we've used here), and `INOUT` (those used to pass parameters to and from stored procedures). The stored procedure code itself is enclosed within `BEGIN` and `END` statements as seen before, and a series of `SELECT` statements are performed to retrieve the values that are then saved into the appropriate variables (by specifying the `INTO` keyword).

Note

Parameter Datatypes The datatypes allowed in stored procedure parameters are the same as those used in tables. Appendix D, "MySQL Datatypes," lists these types.

Note that a recordset is not an allowed type, and so multiple rows and columns could not be returned via a parameter. This is why three parameters (and three `SELECT` statements) are used in the previous example.

To call this updated stored procedure, three variable names must be specified, as seen here:

• Input

```
CALL productpricing(@pricelow,
                    @pricehigh,
                    @priceaverage);
```

• Analysis

As the stored procedure expects three parameters, exactly three parameters must be passed, no more and no less. Therefore, three parameters are passed to this `CALL` statement. These are the names of the three variables that the stored procedure will store the results in.

Note

Variable Names All MySQL variable names must begin with `@`.

When called, this statement does not actually display any data. Rather, it returns variables that can then be displayed (or used in other processing).

To display the retrieved average product price you could do the following:

• Input

```
SELECT @priceaverage;
```

• Output

```
+---------------+
| @priceaverage |
+---------------+
| 16.133571428  |
+---------------+
```

To obtain all three values, you can use the following:

• Input

```
SELECT @pricehigh, @pricelow, @priceaverage;
```

• Output

```
+------------+-----------+---------------+
| @pricehigh | @pricelow | @priceaverage |
+------------+-----------+---------------+
| 55.00      | 2.50      | 16.133571428  |
+------------+-----------+---------------+
```

Here is another example, this time using both `IN` and `OUT` parameters. `ordertotal` accepts an order number and returns the total for that order:

• Input

```
CREATE PROCEDURE ordertotal(
   IN onumber INT,
   OUT ototal DECIMAL(8,2)
)
BEGIN
   SELECT Sum(item_price*quantity)
   FROM orderitems
   WHERE order_num = onumber
   INTO ototal;
END;
```

• Analysis

`onumber` is defined as `IN` because the order number is passed in to the stored procedure. `ototal` is defined as `OUT` because the total is to be returned from the stored procedure. The `SELECT` statement used both of these parameters, the `WHERE` clause uses `onumber` to select the right rows, and `INTO` uses `ototal` to store the calculated total.

To invoke this new stored procedure you can use the following:

• Input

```
CALL ordertotal(20005, @total);
```

• Analysis

Two parameters must be passed to `ordertotal`; the first is the order number and the second is the name of the variable that will contain the calculated total.

To display the total you can then do the following:

• Input

```
SELECT @total;
```

• Output

```
+--------+
| @total |
+--------+
| 149.87 |
+--------+
```

• Analysis

`@total` has already been populated by the `CALL` statement to `ordertotal`, and `SELECT` displays the value it contains.

To obtain a display for the total of another order, you would need to call the stored procedure again, and then redisplay the variable:

• Input

```
CALL ordertotal(20009, @total);
SELECT @total;
```

Building Intelligent Stored Procedures

All of the stored procedures used thus far have basically encapsulated simple MySQL `SELECT` statements. And while they are all valid examples of stored procedures, they really don't do anything more than what you could do with those statements directly (if anything, they just make things a little more complex). The real power of stored procedures is realized when business rules and intelligent processing are included within them.

Consider this scenario. You need to obtain order totals as before, but also need to add sales tax to the total, but only for some customers (perhaps the ones in your own state). Now you need to do several things:

- Obtain the total (as before).
- Conditionally add tax to the total.
- Return the total (with or without tax).

That's a perfect job for a stored procedure:

• Input

```
-- Name: ordertotal
-- Parameters: onumber = order number
--             taxable = 0 if not taxable, 1 if taxable
--             ototal = order total variable

CREATE PROCEDURE ordertotal(
   IN onumber INT,
   IN taxable BOOLEAN,
   OUT ototal DECIMAL(8,2)
) COMMENT 'Obtain order total, optionally adding tax'
BEGIN

   -- Declare variable for total
   DECLARE total DECIMAL(8,2);
   -- Declare tax percentage
   DECLARE taxrate INT DEFAULT 6;

   -- Get the order total
   SELECT Sum(item_price*quantity)
   FROM orderitems
   WHERE order_num = onumber
   INTO total;

   -- Is this taxable?
   IF taxable THEN
      -- Yes, so add taxrate to the total
      SELECT total+(total/100*taxrate) INTO total;
   END IF;

   -- And finally, save to out variable
   SELECT total INTO ototal;

END;
```

• Analysis

The stored procedure has changed dramatically. First of all, comments have been added throughout (preceded by `--`). This is extremely important as stored procedures increase in complexity. An additional parameter has been added`taxable` is a `BOOLEAN` (specify true if taxable, false if not). Within the stored procedure body, two local variables are defined using `DECLARE` statements.

`DECLARE` requires that a variable name and datatype be specified, and also supports optional default values (`taxrate` in this example is set to 6%). The `SELECT` has changed so the result is stored in `total` (the local variable) instead of `ototal`. Then an `IF` statement checks to see if `taxable` is true, and if it is, another `SELECT` statement is used to add the tax to local variable

`total`. And finally, `total` (which might or might not have had tax added) is saved to `ototal` using another `SELECT` statement.

Tip

The `COMMENT` Keyword The stored procedure for this example included a `COMMENT` value in the `CREATE PROCEDURE` statement. This is not required, but if specified, is displayed in `SHOW PROCEDURE STATUS` results.

This is obviously a more sophisticated and powerful stored procedure. To try it out, use the following two statements:

• Input

```
CALL ordertotal(20005, 0, @total);
SELECT @total;
```

• Output

```
+--------+
| @total |
+--------+
| 149.87 |
+--------+
```

• Input

```
CALL ordertotal(20005, 1, @total);
SELECT @total;
```

• Output

```
+---------------+
| @total        |
+---------------+
| 158.862200000 |
+---------------+
```

• Analysis

`BOOLEAN` values may be specified as `1` for true and `0` for false (actually, any non-zero value is considered true and only `0` is considered false). By specifying `0` or `1` in the middle parameter you can conditionally add tax to the order total.

Note

The IF Statement This example showed the basic use of the MySQL IF statement. IF also supports ELSEIF and ELSE clauses (the former also uses a THEN clause, the latter does not). We'll be seeing additional uses of IF (as well as other flow control statements) in future tutorials.

Inspecting Stored Procedures

To display the CREATE statement used to create a stored procedure, use the SHOW CREATE PROCEDURE statement:

• Input

```
SHOW CREATE PROCEDURE ordertotal;
```

To obtain a list of stored procedures including details on when and who created them, use SHOW PROCEDURE STATUS.

Note

Limiting Procedure Status Results SHOW PROCEDURE STATUS lists all stored procedures. To restrict the output you can use LIKE to specify a filter pattern, for example:

```
SHOW PROCEDURE STATUS LIKE 'ordertotal';
```