# Database Management System

Mc Joshua de Lima

Bachelor of Science in Information Technology
College of Computing Studies

# Table of Contents

# List of Tables

# MODULE 5
# STORED PROCEDURES

## Introduction

MySQL has long been criticized for its lack of stored procedures by application developers, database administrators, business analysts, and rival databases. With the announcement of version 5.0 came the news that stored procedures are now an option for users of MySQL. This new functionality has generated some excitement in the MySQL community, as well as in the outlying database market. MySQL users who have wanted the ability to use stored procedures but are living with MySQL for other reasons are rejoicing. Likewise, folks who are using other databases because of a need for stored procedures are reevaluating MySQL as a possible alternative to their current database choice (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## Learning Outcomes

At the end of this module, students should be able to:

1. Use stored procedures in managing database.
2. Use nested stored procedures to produce the desired outcome.
3. Use operators in MySQL statements.

## Lesson 1. Stored Procedure Considerations

A stored procedure is a collection of SQL statements used together. Stored procedures allow you to go beyond the typical single-statement database query used to retrieve a set of records or update a row. Stored procedure syntax supports variables, conditions, flow controls, and cursors, so a stored procedure can perform complex processing within the database between the database call and the resulting return. Stored procedures can consist of as little as one statement, or they may contain hundreds or even thousands of lines.

With the ability to store complex processing inside the database, application developers and database administrators find that, in certain cases, they prefer to have some of the processing logic performed in the database before the data is returned to the client or application. Putting logic into the database is a heavily debated topic, and should not be done without first considering the advantages and disadvantages of doing so (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

**Stored Procedure Advantages**

- Stored procedures allow you to combine multiple queries into a single trip to the database. This means you can reduce traffic between the client and the database by not needing to make multiple requests for multiple actions. Depending on your application, saving network traffic can be significant enough to offset any counter argument.
- If you are grabbing many rows from the database and using business logic to limit the results, a stored procedure may be able to encapsulate that logic and reduce the amount of data returned to the client or application, as well as reduce the processing needed in the application before presenting the results.
- Stored procedures provide a clean interface to the data. Rather than needing to build a query in your code, you can reduce your SQL to a single, simple call statement.
- In some cases, having your SQL stored in the database makes managing queries much easier, especially in distributed or compiled systems, where deploying new codewith embedded queries is very difficult.
- Keeping queries in the database allows the database administrator control over the creation and optimization of queries, tables, and indexes used in returning data to the calling client. The database administrator has a deeper understanding of the data model, relationships, and performance of the database. SQL statements coming from the database administrator will be optimized with the data model in mind, and perhapswill spur tweaks in the database configuration and indexes.
- Stored procedures abstract the structure of the data from the application, which can help when changes are needed in the data model by having a central location for all query changes. In programming, this is referred to as class or package encapsulation. The stored procedures represent a black box, where developers don’t need to be

familiar with the internals of the procedure. As long as developers know how to call the procedure and how the results will come back, they can be ignorant about the procedure details.

- Where security is important, stored procedures allow controlled access to viewing and changing data. Callers do not need to have permission on specific tables; if their query can be encapsulated in a stored procedure, they need only the ability to access that procedure.
- Using stored procedures means you are passing a single parameter or a small list of parameters to the database. Compared with building SQL statements in your application to pass to the database, the parameterized call is more secure. If you're building SQL on the fly in your code, you might be more open to attacks like SQL injection, where attackers attempt to spoof your program and alter your dynamically built query to expose, change, or destroy your data.
- Some databases optimize the statements of your stored procedure, parsing and organizing the pieces of the procedure when it is created. This reduces the amount of work necessary when the procedure is called, meaning that queries and operations in the procedure are faster than if they were sent from the client to parse and process atrun time.

**Other Considerations in Using Stored Procedures**

Before we move into the implementation details for stored procedures in MySQL, you should be thinking about how stored procedures will fit into your application design. As you look at the possibilities for moving pieces of your application into the database, consider how you will draw lines between layers in your application. Perhaps you want just a few procedures to run some complex data manipulation, or maybe you are thinking about creating a completedata abstraction layer.

If you plan on creating more than one or two procedures, you should also be thinking about how you'll break the functionality of your procedures into small, reusable chunks. You may also consider creating a style guide and best-practices document to unify the interfaces and internals of your procedures.

## Lesson 2. Stored Procedures in MySQL

Database vendors use a variety of programming languages and syntax for building and managing stored procedures. Many of these databases share a set of core SQL commands, but most of them have added extensions to facilitate the level of complexity that can be attained within the stored procedure. Oracle procedures are written in PL/SQL. Microsoft SQL Server 2000 procedures are written in Transact-SQL (T-SQL). To write procedures for PostgreSQL, you use PL/psSQL. Each implementation includes some common commands, and then an extended syntax for accomplishing more advanced logic.

MySQL developers have taken the expected approach in their implementation of stored procedures. A database focused on simplicity and maximum performance would likely implement a simple set of features that supply the most amount of control to users wanting to move logic into the database. MySQL has done this by implementing the SQL:2003 standard for stored procedures and has added minimal MySQL-specific syntax. In the cases where MySQL provides an extended use of a statement, the MySQL documentation (and this book) notes the extension to the standard.

The SQL:2003 standard provides a basic set of commands for building multiple-statement interactions with the database. SQL:2003 was published in 2003 as the replacement for the previous SQL standard, SQL:1999. These standards include specifications for syntax and behavior for SQL commands that are used to build, create, and maintain stored procedures. MySQL's choice to stick to the SQL:2003 standard means that stored procedures created in MySQL can be seamlessly used in other databases that support this standard. Currently, IBM's DB2 and Oracle Database 10g are compliant with SQL:2003. The success of moving a stored procedure from Oracle or DB2 into MySQL will depend on whether any of the vendor extensions have been used. Even if the vendor supports SQL:2003, if a stored procedure uses vendor-specific syntax, MySQL will fail on an unrecognized command when attempting to create the procedure.

The MySQL implementation provides a wide array of controls for processing data and logic in the database. It doesn't have the extended syntax bells and whistles of other database systems, but it does provide a rich set of basic commands that can create some incrediblypowerful procedures.

Stored procedures are processed by the MySQL server, and they are independent of the storage engine used to store your data. If you use a feature of a particular storage engine in your stored procedure statement, you will need to continue to use that table type to use the stored procedure. MySQL stores the data for stored procedures in the proc table in the mysql database. Even though procedures are all stored in one place, they are created and called by either using the current database or by prepending a database name onto the various procedure statements(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## Lesson 3. Building Stored Procedures

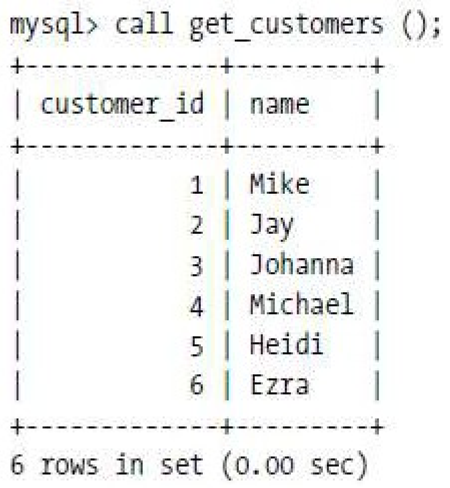
SQL:2003 sets forth a set of commands to create procedures; declare variables, handlers, and conditions; and set up cursors and constructs for flow control. In its simplest form, you can create a stored procedure with a CREATE statement, procedure name, and a single SQL statement. Listing 5.1 shows just how simple this can be(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

**Listing 5.1 Creating a Single-Statement Procedure**

```
mysql> create procedure get_customers ()
SELECT customer_id,name FROM customer;
```

However frivolous Listing 5.1 may appear, it contains the required parts: a CREATE statement with a procedure name and a SQL statement. Calling the stored procedure to get the results is simple, as demonstrated in Listing 5.2.

**Listing 5.2. Calling a Single-Statement Procedure**



```
mysql> call get_customers ();
+-------------+---------+
| customer_id | name    |
+-------------+---------+
|           1 | Mike    |
|           2 | Jay     |
|           3 | Johanna |
|           4 | Michael |
|           5 | Heidi   |
|           6 | Ezra    |
+-------------+---------+
6 rows in set (0.00 sec)
```

Other than abstracting the syntax of the query from the caller, this example doesn't really justify creating a procedure. The same result is just as easily available with a single query from your application.

As a more realistic example, let's consider the scenario of merging duplicate accounts in your online ordering system. Your online store allows a user to create an account, with a userdefined login and password, to use for placing orders. Suppose user Mike places an order or two, and then doesn't visit your site for a while. Then he returns and signs up again, inadvertently creating a second account. He places a few more orders. At some point, he realizes that he has two accounts and puts in a request to have the old account removed. He says that he would prefer to keep all the old orders on the newer account.

This means that in your database, you'll need to find all the information associated with the old account, move it into the new account, and delete the old account. The new account record probably has core pieces of information like name, address, and phone, which won't need to change. The data to be moved may include address book and payment information, as well as Mike's orders. Anywhere in your system where a table has a relationship with your customer, you'll need to make a change. Of course, you should check for the existence of the accounts, and the employee who makes that change may want to have a report of how many records were changed.

Creating the series of statements to process this data merge in your code is possible, but using a procedure to handle it would simplify your application. Listing 5.3 demonstrates how a stored procedure might solve the requirements of this merge account request.

**Listing 5.3 Creating a Multistatement Stored Procedure**

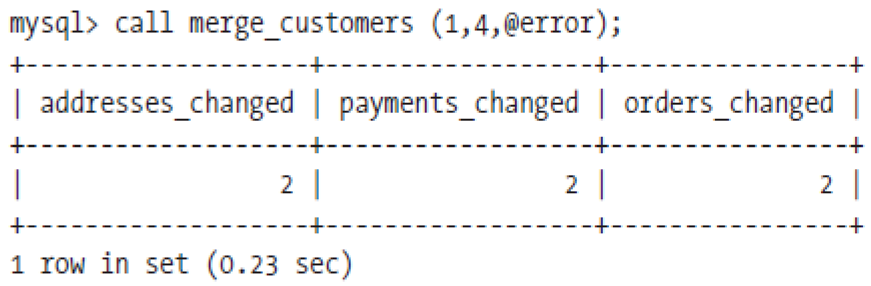
```
DELIMITER //
CREATE PROCEDURE merge_customers
(IN old_id INT, IN new_id INT, OUT error VARCHAR(100))
SQL SECURITY DEFINER
COMMENT 'merge customer accounts'
BEGIN
        DECLARE old_count INT DEFAULT 0;
        DECLARE new_count INT DEFAULT 0;
        DECLARE addresses_changed INT DEFAULT 0;
        DECLARE payments_changed INT DEFAULT 0;
        DECLARE orders_changed INT DEFAULT 0;
        ## check to make sure the old_id and new_id exists
        SELECT count(*) INTO old_count FROM customer WHERE customer_id = old_id;
        SELECT count(*) INTO new_count FROM customer WHERE customer_id = new_id;
        IF !old_count THEN
                SET error = 'old id does not exist';
        ELSEIF !new_count THEN
                SET error = 'new id does not exist';
        ELSE
                UPDATE address SET customer_id = new_id WHERE customer_id = old_id;
                SELECT row_count() INTO addresses_changed;
                UPDATE payment SET customer_id = new_id WHERE customer_id = old_id;
                SELECT row_count() INTO payments_changed;
                UPDATE cust_order SET customer_id = new_id WHERE customer_id = old_id;
                SELECT row_count() INTO orders_changed;
                DELETE FROM customer WHERE customer_id = old_id;
                SELECT addresses_changed,payments_changed,orders_changed;
        END IF;
END
//
DELIMITER ;
```

When entering multiple statement blocks into MySQL, you need to first change the default delimiter to something other than a semicolon (;), so MySQL will allow you to enter a ; without having the client process the input. Listing 5.3 begins by using the delimiter statement:

DELIMITER //, which changes the delimiter to //. When you're ready to have your procedure created, type //, and the client will process your entire procedure. When you're finished working on your procedures, change the delimiter back to the standard semicolon with: DELIMITER ;, as you can see at the end of Listing 5.3.

Listing 5.4 shows how to call this procedure with the required parameters and get the results from the procedure. We'll look at the details of executing stored procedures in the "Using Stored Procedures" section later in this chapter.

**Listing 5.4 Calling the Stored Procedure**



```
mysql> call merge_customers (1,4,@error);
+-------------------+------------------+----------------+
| addresses_changed | payments_changed | orders_changed |
+-------------------+------------------+----------------+
|                 2 |                2 |              2 |
+-------------------+------------------+----------------+
1 row in set (0.23 sec)
```

**The CREATE Statement**

You create a stored procedure using the CREATE statement, which takes a procedure name, followed by parameters in parentheses, followed by procedure characteristics, and ending with the series of statements to be run when the procedure is called. Here is the syntax:

```
mysql> CREATE PROCEDURE [database.]<name> ([<parameters>]) [<characteristics>]

<body statements>
```

The name may be prefixed with a database name, and it must be followed by parentheses. If the database is not provided, MySQL creates the procedure in the current database or gives a No database selected error if a database is not active. Procedure names can be up to 64 characters long.

You can set parameters for a stored procedure using the following syntax:

```
[IN|OUT|INOUT] <name> <data type>
```

If you don't specify IN, OUT, or INOUT for the parameter, it will default to IN. These three types of parameters work as follows:

- An IN parameter is set and passed into the stored procedure to use internally in its processing.
- An OUT parameter is set within the procedure, but accessed by the caller.
- An INOUT parameter is passed into the procedure for internal use, but is also availableto the caller after the procedure has completed.

The name and data type of the parameter are used in the stored procedure for referencing and setting values going in and out of the procedure. The data type can be any valid data type for MySQL, and it specifies what type of data will be stored in the parameter.

The stored procedure characteristics include a number of options for how the stored procedure behaves. Table 5.1 lists the available options with a description of how they affect the stored procedure.

**Table 5.1 Characteristics Used to Create a Stored Procedure**

| **Characteristic** | **Value** | **Description** |
|---|---|---|
| LANGUAGE | SQL | This is the language that was used to write the stored procedure. While MySQL intends to implement other languages with external procedures, currently SQL is the only valid option. |
| SQL SECURITY | DEFINER or INVOKER | to use for permissions when running the procedure. If it's set to DEFINER, the stored procedure will be run using the privileges of the user who created the procedure. If INVOKER is specified, the user calling the procedure will be used for obtaining access to the tables. The default, if not specified, is |

|  |  |  |
|---|---|---|
|  |  | DEFINER. |
| COMMENT |  | The COMMENT characteristic is a place to enter notes about a stored procedure. The comment is displayed in SHOW CREATE PROCEDURE commands |

## The Procedure Body

The body of a stored procedure contains the collection of SQL statements that make up the actual procedure. In addition to the typical SQL statements, you use to interact with data in your database, the SQL:2003 specification includes a number of additional commandsto store variables, make decisions, and loop over sets of records.

## BEGIN and END Statements

You use the BEGIN and END statements to group statements in procedures with more than one SQL statement. Declarations can be made only within a BEGIN . . . END block.

You can define a label for the block to clarify your code, as shown here:

```
customer: BEGIN
<SQL statement>;
<SQL statement>;
END customer
```

The labels must match exactly.

## The DECLARE Statement

The DECLARE statement is used to create local variables, conditions, handlers, and cursors within the procedure. You can use DECLARE only as the first statements immediately within a BEGIN block. The declarations must occur with variables first, cursors second, and handlers last. A common declaration is the local variable, which is done with a variable name and type:

```
DECLARE <name> <data type> [DEFAULT];
```

Variable declarations can use any valid data type for MySQL, and may include an optional default value. In Listing 5.3, several declarations are made, including a number of variables for counting items as the statements in the procedure are processed:

```
DECLARE new_count INT DEFAULT 0;
```

**Variables**

Stored procedures can access and set local, session, and global variables. Local variables are either passed in as parameters or created using the DECLARE statement, and they are used in the stored procedure by referencing the name of the parameter or declared variable. You can set variables in several ways. Using the DECLARE statement with a DEFAULT will set the value of a local variable:

```
DECLARE customer_count INT DEFAULT 0;
```

You can assign values to local, session, and global variables using the SET statement:

```
SET customer_count = 5;
```

MySQL's SET statement includes an extension to the SQL:2003 standard that permits setting multiple variables in one statement:

```
SET customer_count = 5, order_count = 50;
```

Using SELECT . . . INTO is another method for setting variables within your stored procedure. This allows you to query a table and push the results into a variable as a part ofthe query. SELECT . . . INTO works only if you are selecting a single row of data:

```
SELECT COUNT(*) INTO customer_count FROM customer;
```

You can also select multiple values into multiple variables:

```
SELECT customer_id,name INTO new_id,new_name FROM customer LIMIT 1;
```

**Conditions and Handlers**

When making declarations in your stored procedure, your list of declarations can include statements to indicate special handling when certain conditions arise. When you have a collection of statements being processed, being able to detect the outcome of those statements and proactively do something to help the procedure be successful can be important to your caller.

Suppose one of the stored procedures created for your online store included a statement to update the customer's name. The column for the customer's name is CHAR(10), which is smaller than you would like, but is the most your legacy order-entry system can handle. The normal behavior for MySQL when updating a record is to truncate the inserted value to a length that fits the column. For numerous reasons, this is unacceptable to you. Fortunately, when MySQL does a truncation, it issues a warning and returns an error, and also sets the SQLSTATE to indicate that during the query, the data was truncated.

Handlers are designed to detect when certain errors or warnings have been triggered by statements and allow you to take action. A handler is declared with a handler type, condition, and statement:

```
DECLARE <handler type> HANDLER FOR <condition> <statement>;
```

**Handler Types**

The handler type is either CONTINUE or EXIT.3 CONTINUE means that when a certain error or warning is issued, MySQL will run the provided statement and continue running the statements in the procedure. The EXIT handler type tells MySQL that when the condition is met, it should run the statement and exit the current BEGIN . . . END block.

Here's a handler statement with an EXIT handler type:

```
DECLARE EXIT HANDLER FOR truncated_name UPDATE customer SET name = old_name WHERE
customer_id = cust_id;
```

In this statement, the EXIT handler type tells the procedure to execute the statement, and theexit when a truncation occurs.

**Conditions**

The handler condition is what triggers the handler to act. You can define your own conditions and reference them by name, or choose from a set of conditions that are provided by default in MySQL. Table 5.2 shows the MySQL handler conditions.

**Table 5.2.MySQL Handler Conditions**

| **Condition** | **Description** |
|---|---|
| SQLSTATE '<number>' | A specific warning or error number, which is described in the MySQL documentation. The number must be enclosed in quotes (typically single). |
| <self-defined condition name> | The name of the self-defined condition you created using the DECLARE . . . CONDITION statement. |
| SQLWARNING | Matches any SQLSTATE that begins with 01. Using this condition will allow you to catch a wide range of states. |

| | |
|---|---|
| NOT FOUND | Matches any SQLSTATE beginning with 02. Using thisstate lets you catch any instance where the query references a missing table, database, and so on. |
| SQLEXCEPTION | Matches every SQLSTATE except those beginning with 01 or 02. |
| <MySQL error> | Using a specific error will cause the handler to execute for the specific MySQL error. |

To create a self-defined condition, use a condition declaration with a name and a value:

```
DECLARE <condition name> CONDITION FOR <condition value>;
```

The condition name will be used in a DECLARE . . . HANDLER definition. The condition value can be either a MySQL error number or a SQLSTATE code. For example, to catch when some data has been truncated, the condition declaration with the MySQL error number looks like this:

```
DECLARE truncated_name CONDITION FOR 1265;
```

Or if you wanted to use the SQLSTATE number, you would write the same statement like this:

```
DECLARE truncated_name CONDITION FOR SQLSTATE '01000';
```

**Statements**

The last piece of the handler declaration is a statement, which will be run before the stored procedure either continues or exits, depending on the handler type you chose. For example, to catch a case where the name had been truncated, your stored procedure might look like the one shown in Listing 5.5.

```
DELIMITER //
CREATE PROCEDURE update_name (IN cust_id INT, IN new_name VARCHAR(20))
BEGIN
        DECLARE old_name VARCHAR(10);
        DECLARE truncated_name CONDITION for 1265;
        DECLARE EXIT HANDLER FOR truncated_name
                UPDATE customer SET name = old_name WHERE customer_id = cust_id;
        SELECT name INTO old_name FROM customer WHERE customer_id = cust_id;
        UPDATE customer SET name = new_name WHERE customer_id = cust_id;

        SELECT customer_id,name FROM customer WHERE customer_id = cust_id;
END
//
DELIMITER ;
```

The update_name procedure accepts a customer ID (cust_id) and a new name (new_name). The first two statements declare a variable to store the old name and a condition named truncated_name, which specifies MySQL error 1265 as the condition. MySQL error 1265 indicates that a field in the statement was truncated when the statement was processed. The third declaration is a handler statement that tells the procedure that if the truncated_namestate is reached, to update the customer record to the old name and exit.

The stored procedure runs the declarations first, and then selects the current name for that customer into the old_name variable. On the following UPDATE statement, depending on the length of the name to be inserted, the query result may be a MySQL error 1265. If so, thehandler for truncated_name runs the statement associated with the handler:

```
UPDATE customer SET name = old_name WHERE customer_id = cust_id;
```

This query sets the name back to the original value. The procedure then exits, and no record is returned to the client.

## Flow Controls

SQL:2003 flow constructs give you a number of statements to control and organize your statement processing. MySQL supports IF, CASE, LOOP, LEAVE, ITERATE, REPEAT, and WHILE, but does not currently support the FOR statement.

### IF

The IF statement behaves as you would expect if you've written code in another language. It checks a condition, running the statements in the block if the condition is true. You can add ELSEIF statements to continue attempting to match conditions and also, if desired, include a final ELSE statement. Listing 5.6 shows a piece of a procedure where the shipping cost is being calculated based on the number of days the customer is willing to waitfor delivery. delivery_day is an integer parameter passed into the procedure.

**Listing 5.6. IF Statement**

```
IF delivery_day = 1 THEN
        SET shipping = 20;
ELSEIF delivery_day = 2 THEN
        SET shipping = 15;
ELSEIF delivery_day = 3 THEN
        SET shipping = 10;
ELSE
        SET shipping = 5;
END IF;
```

### CASE

If you're checking a uniform condition, such a continual check for number of shipping days, you might be better off using the CASE construct. Listing 5.7 shows how this same logic demonstrated in Listing 9-6 could be processed using the CASE statement. Not only does it seem to improve the readability of the code, but the code in Listing 5.7 runs at least twice as fast as the code in Listing 5.6. delivery day is an integer parameter passed into the procedure

**Listing 5.7 CASE Statement**

```
CASE delivery_day
WHEN 1 THEN
        SET shipping = 20;
WHEN 2 THEN
        SET shipping = 15;
WHEN 3 THEN
        SET shipping = 10;
ELSE
        SET shipping = 5;
END case;
```

The CASE control can also operate without an initial case value, evaluating a condition on each WHEN block. Listing 9-8 shows the shipping calculator using this syntax. As with Listing 5.7, Listing 5.8 runs significantly faster than the IF-based logic in Listing 5.6.

**Listing 5.8. CASE Statement with Condition Checks**

```
CASE
WHEN delivery_day = 1 THEN
        SET shipping = 20;
WHEN delivery_day = 2 THEN
        SET shipping = 15;
WHEN delivery_day = 3 THEN
        SET shipping = 10;
ELSE
        SET shipping = 5;
END CASE;
```

Now that you are up to speed with checking values, we'll turn our attention to the constructs for repeating. The LOOP, LEAVE, ITERATE, REPEAT, and WHILE statements provide methods to work through a given number of conditions.

## LOOP and LEAVE

The LOOP statement creates an ongoing loop that will run until the LEAVE statement is invoked. Optional to the LOOP is a label, which is a name and a colon prefixed to the LOOP statement, with the identical name appended to the END LOOP statement. Listing 5.9 demonstrates a LOOP and LEAVE construct.

**Listing 5.9. LOOP Statement with LEAVE**

```
increment: LOOP
SET count = count + 1;
IF count > in_count THEN LEAVE increment;
END IF;
END LOOP increment;
```

The LEAVE statement is designed to exit from any flow control. The LEAVE statement must be accompanied by a label.

## ITERATE

You can use ITERATE in a LOOP, WHILE, or REPEAT control to indicate that the control should iterate through the statements in the loop again. Listing 5.10 shows ITERATE added to the increment example in Listing 5.9. Adding the IF condition to check if the count is less than 20, and if so iterating, means that the value of count, when the loop is complete, will never be less than 20, because the ITERATE statement ensures that the addition statementis run repeatedly until the count reaches 20.

**Listing 5.10. Loop with ITERATE Statement**

```
DELIMITER //
CREATE PROCEDURE increment (IN in_count INT)
BEGIN
DECLARE count INT default 0;

increment: LOOP
SET count = count + 1;
IF count < 20 THEN ITERATE increment; END IF;
IF count > in_count THEN LEAVE increment;
END IF;
END LOOP increment;

SELECT count;
END
//
DELIMITER ;
```

## WHILE

The WHILE statement is another mechanism to loop over a set of statements until a condition is true. Unlike LOOP, where the condition is met within the loop, the WHILE statement requires specification of the condition when defining the statement. As with loops, you can add a name to give a name to the WHILE construct. Listing 5.11 shows a simple use of this statement.

**Listing 5.11. WHILE Statement**

```
WHILE count < 10 DO
SET count = count + 1;
END WHILE;
```

### REPEAT

To loop over a set of statements until a post-statement condition is met, use the REPEAT statement. Listing 5.12 shows a simple use. The check_count label is optional, as isthe label with other constructs.

**Listing 5.12. REPEAT Statement**

```
check_count:  REPEAT
SET count = count + 1;
UNTIL count > 10
END REPEAT check_count;
```

## Lesson 4. Using Stored Procedures

If you've gone to all the trouble of creating a procedure, you probably want to put it to use. You may be calling the procedures directly from the MySQL command-line client or from a program written in PHP, Java, Perl, Python, or another language. Here, we'll look at how to call procedures from the command line and from PHP, just to demonstrate calling procedures from a program. Check the documentation for the specific language you're using to see which drivers are needed and how the interface for procedures and parameters works in that language(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

### Calling Procedures from the MySQL Client

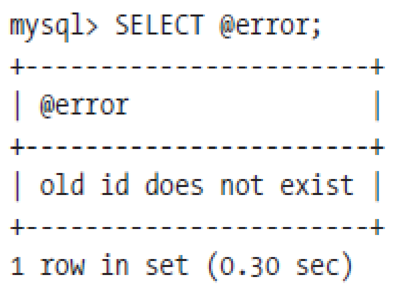
From the MySQL client, you use the CALL statement to execute a procedure, providingthe procedure name and correct number of arguments.

```
CALL [database.]<procedure name> ([<parameter>, <parameter>, …]);
```

Calling a simple procedure without any parameters is fairly straightforward, as you saw earlier, when we demonstrated calling the get_customer procedure (Listing 5.2). Listing 5.13 shows an example of calling a stored procedure that requires three arguments: an old customer ID as the first IN argument, a new customer ID as the second IN argument, and an OUT argument used in the procedure for setting an error message. Once the stored procedure has been executed, the @error variable contains a string set inside the stored procedure.

**Listing 5.13. Calling a Stored Procedure with IN and OUT Parameters**

```
mysql> CALL merge_customers (8,9,@error);
Query OK, 0 rows affected (0.01 sec)
```



```
mysql> SELECT @error;
+-----------------------+
| @error                |
+-----------------------+
| old id does not exist |
+-----------------------+
1 row in set (0.30 sec)
```

If you call a procedure with the wrong number of arguments, MySQL gives an error:

```
mysql> CALL shop.merge_customers (1,2);
ERROR 1318 (42000): Incorrect number of arguments for PROCEDURE merge_customers;
\expected 3, got 2
```

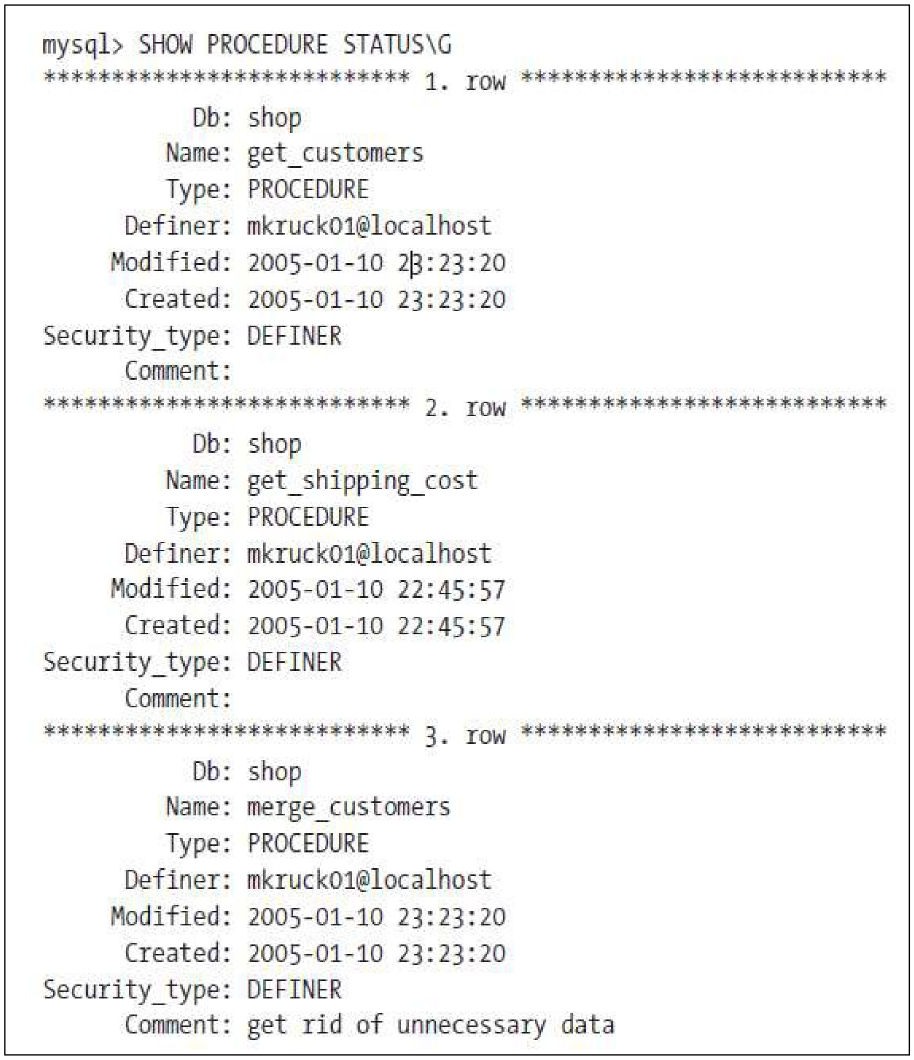
## Managing Stored Procedures

Most of the work in an environment where stored procedures are used is in creating the stored procedure. However, at some point, you will need to manage the procedures in your database. MySQL provides a set of commands for this purpose

### Viewing Stored Procedures

You have several options when viewing information about stored procedures. To get a summary of the procedures across all databases in your system, use SHOW PROCEDURE STATUS, which will give you a summary of information about all the stored procedures in your system. Listing 5.13 shows the output for three procedures used for listings in this chapter. Using the \G option outputs in rows instead of columns.

**Listing 5.13. Output of SHOW PROCEDURE STATUS**



```
mysql> SHOW PROCEDURE STATUS\G
*************************** 1. row ***************************
           Db: shop
         Name: get_customers
         Type: PROCEDURE
      Definer: mkruck01@localhost
     Modified: 2005-01-10 23:23:20
      Created: 2005-01-10 23:23:20
Security_type: DEFINER
      Comment:
*************************** 2. row ***************************
           Db: shop
         Name: get_shipping_cost
         Type: PROCEDURE
      Definer: mkruck01@localhost
     Modified: 2005-01-10 22:45:57
      Created: 2005-01-10 22:45:57
Security_type: DEFINER
      Comment:
*************************** 3. row ***************************
           Db: shop
         Name: merge_customers
         Type: PROCEDURE
      Definer: mkruck01@localhost
     Modified: 2005-01-10 23:23:20
      Created: 2005-01-10 23:23:20
Security_type: DEFINER
      Comment: get rid of unnecessary data
```

This command can be limited by appending a LIKE clause, in this case limiting the output to just returning the merge_customer procedure.

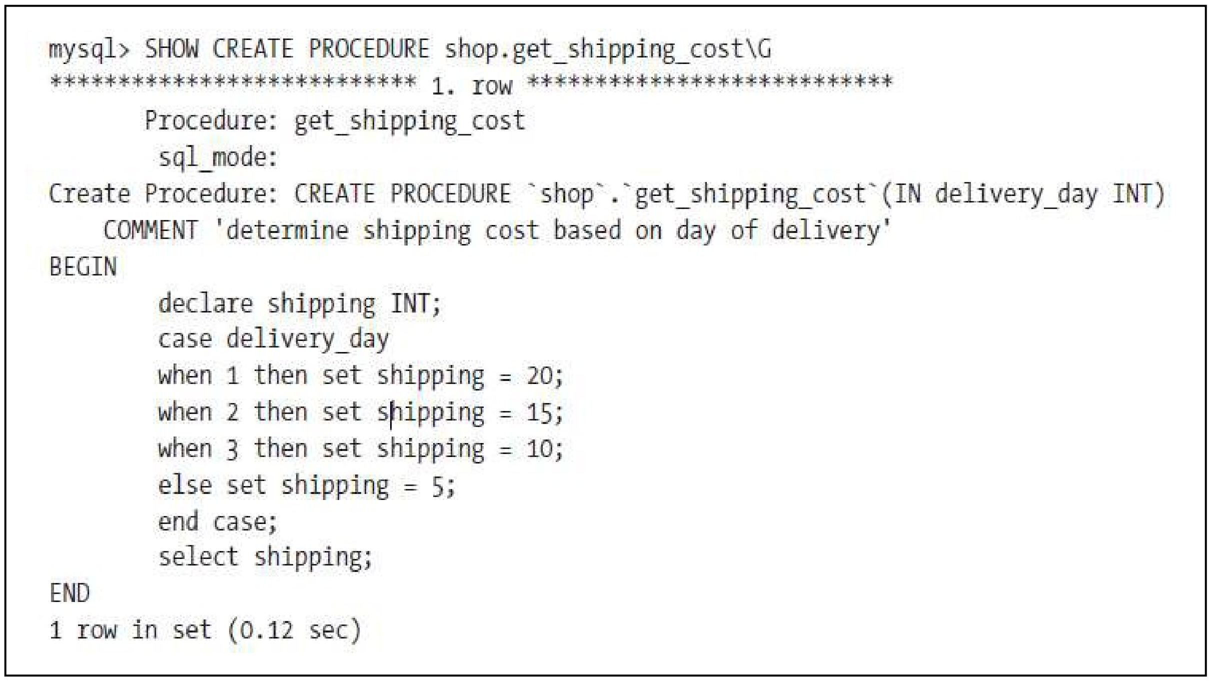
```
mysql> SHOW PROCEDURE STATUS LIKE 'merge%'\G
```

The SHOW PROCEDURE STATUS statement gives you a nice summary view of all the procedures in the databases on your machine. To get more details on a stored procedure,use the SHOW CREATE PROCEDURE statement:

```
SHOW CREATE PROCEDURE [<database>.]<procedure name>;
```

This statement shows you the name and the CREATE statement. Listing 5.14 showsan example of the output for the get_shipping_cost procedure.

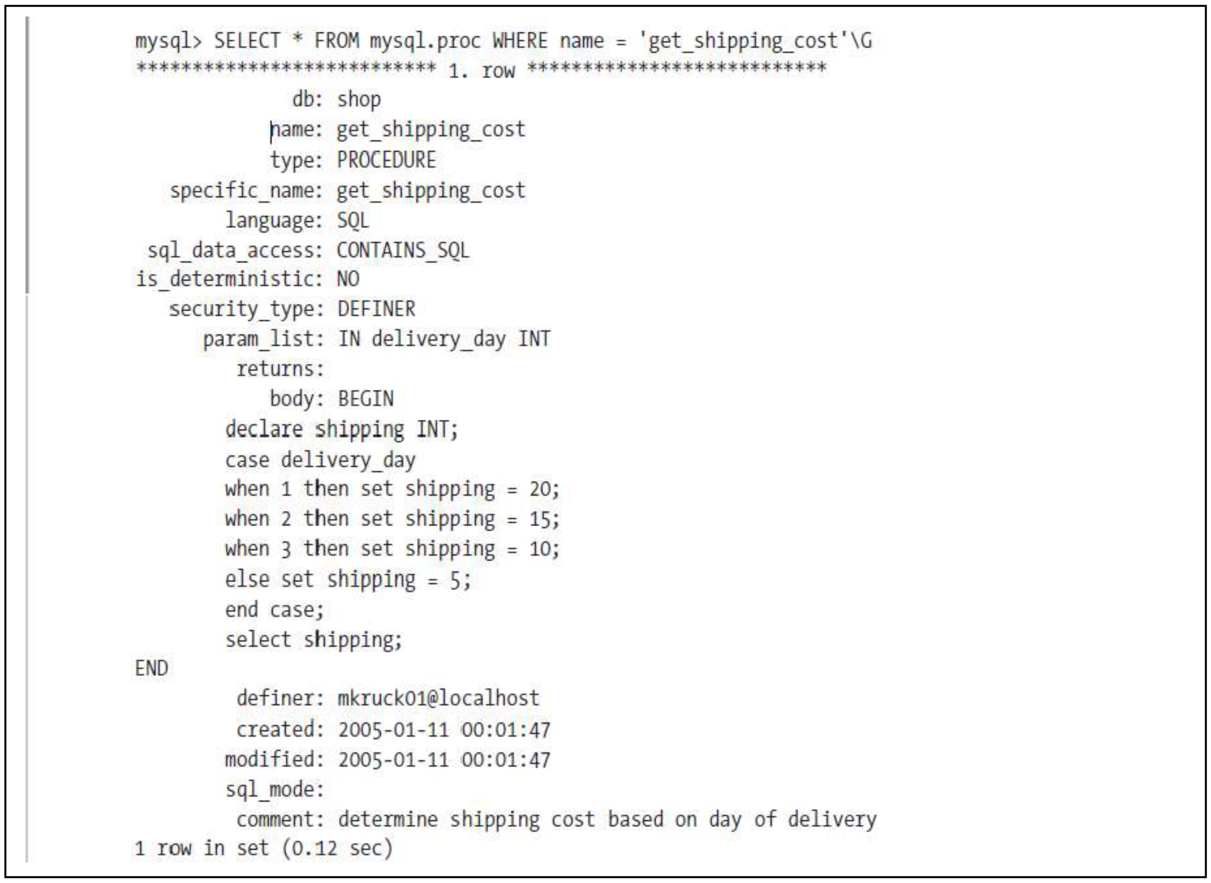
**Listing 5.14. Output of SHOW CREATE PROCEDURE**



```
mysql> SHOW CREATE PROCEDURE shop.get_shipping_cost\G
************************** 1. row **************************
       Procedure: get_shipping_cost
        sql_mode:
Create Procedure: CREATE PROCEDURE `shop`.`get_shipping_cost`(IN delivery_day INT)
    COMMENT 'determine shipping cost based on day of delivery'
BEGIN
        declare shipping INT;
        case delivery_day
        when 1 then set shipping = 20;
        when 2 then set shipping = 15;
        when 3 then set shipping = 10;
        else set shipping = 5;
        end case;
        select shipping;
END
1 row in set (0.12 sec)
```

Neither of the views we've discussed thus far shows you everything there is to know about a procedure. The summary provides only a few pieces of summary information, and SHOW ↬ CREATE PROCEDURE shows the name, along with the body as a large, unreadable CREATE statement. If you have SELECT access on the proc table in the mysql database, a SELECT statement will show you everything there is to know about all procedures or a particular procedure. Listing 5.15 shows the output from a SELECT of the get_shipping_cost procedure, which shows the procedure's database, name, language, security type, parameter list, body, definer, comment, and other information.

**Listing 5.15. Output of SELECT from the mysql.proc Table**



```
mysql> SELECT * FROM mysql.proc WHERE name = 'get_shipping_cost'\G
*************************** 1. row ***************************
              db: shop
            name: get_shipping_cost
            type: PROCEDURE
   specific_name: get_shipping_cost
        language: SQL
 sql_data_access: CONTAINS_SQL
is_deterministic: NO
   security_type: DEFINER
      param_list: IN delivery_day INT
         returns:
            body: BEGIN
        declare shipping INT;
        case delivery_day
        when 1 then set shipping = 20;
        when 2 then set shipping = 15;
        when 3 then set shipping = 10;
        else set shipping = 5;
        end case;
        select shipping;
END
         definer: mkruck01@localhost
         created: 2005-01-11 00:01:47
        modified: 2005-01-11 00:01:47
        sql_mode:
         comment: determine shipping cost based on day of delivery
1 row in set (0.12 sec)
```

As you can see, if you want to view everything there is to know about a procedure, the direct SELECT on the mysql.proc table will provide the most information(Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## Lesson 5. Stored Procedure Permissions

For permissions to create and call stored procedures, MySQL relies on the existing permissions scheme, which is covered in Chapter 15. Specific to procedures, the MySQL permissions scheme has the CREATE ROUTINE, ALTER ROUTINE, and EXECUTE privilege.

The permissions required for working with stored procedures are as follows:

- Viewing permissions: To view stored procedures with SHOW PROCEDURE STATUS,you must have SELECT access to the mysql.proc table. To be able to use the SHOW

CREATE PROCEDURE, you must have either SELECT access to the mysql.proctable or the ALTER ROUTINE privilege for that particular procedure.

- Calling permissions: To call a stored procedure, you need the ability to connect to the server and have the EXECUTE permission for the procedure. EXECUTE permissions can be granted globally (in the mysql.user table), at the database level (in the mysql.dbtable), or for a specific routine (in the mysql.procs_priv table).
- Creating and altering permissions: To govern creating and altering a stored procedure, MySQL uses the CREATE ROUTINE and ALTER ROUTINE privilege. As with the EXECUTE privilege, permissions for creating or changing procedures can be granted globally (in the mysql.user table), at the database level (in the mysql.db table), or fora specific routine (in the mysql.procs_priv table).
- Dropping permissions: To drop a procedure, you must have the ALTER ROUTINE privilege. Permissions for dropping procedures can be granted globally (in the mysql.user table), at the database level (in the mysql.db table), or for a specific routine (in the mysql. procs_priv table).(Ramakrishhan, Raghu and Gehrke,Johannes,2018)

The success of a stored procedure call is also affected by the procedure's SQL SECURITY characteristic. If set to DEFINER, the procedure will be run with the permissions of the user who created the procedure. Procedures will be run as the calling user if SQL SECURITY is set to INVOKER. In either case, the INVOKER or DEFINER must have appropriate access to the tables used in the stored procedure or calling the procedure will result in a permission error.

Having the option to run procedures with the permissions of the creator means that you can create a set of procedures by a user with access to all of the tables, and allow a user who has no permissions in the tables but does have the ability to connect to the server and execute the procedure, to run it. This can be a simple, but excellent, way to simplify and enforce security in your database.

## Assessment Task

Based from your understanding, explain what is Stored Procedures and the benefits of using stored procedures. (2-3 Paragraph only)

**Note: Your answer will be tested with Plagiarism checker.**

## Summary

Stored procedures in MySQL are a welcome and exciting addition to the 5.0 release. While there's a lot of power, and perhaps some efficiency, in moving logic into your database, it's important to consider if and how procedures fit into your existing application. Hasty decisions based on excitement to use cool technology usually lead to problems down the road. As mentioned in the chapter, users should exercise caution in adopting the stored procedure functionality until the stability of the 5.0 server matches their environment requirements. For most users, waiting for the stable release is probably the best choice. MySQL's choice of SQL:2003 provides a good set of statements for developing procedures and a standard for potential inter-database procedure exchange. MySQL provides a good setof tools for creating, altering, dropping, and viewing procedures.

## References

Ramakrishhan, Raghu and Gehrke,Johannes (2018) ® Database Management Systems. McGraw Hill

(Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Source :SQL for beginners: A guide to study SQL programming and database management systems

Database management systemsGupta, G. K.,2020

# MODULE 6
# USING STORED ROUTINES, TRIGGERS, AND EVENTS

## Introduction

A stored function performs a calculation and returns a value that can be used in expressions just like a built-in function such as RAND( ), NOW( ), or LEFT( ). A stored procedure performs calculations for which no return value is needed. Procedures are not used in expressions; they are invoked with the CALL statement. A procedure might be executed to update rows in a table or produce a result set that is sent to the client program. One reason for using a stored routine is that it encapsulates the code for performing a calculation. This enables you to perform the calculation easily by invoking the routine rather than by repeatingall its code each time (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## Learning Outcomes

At the end of this module, students should be able to:

1. Create compound statements.
2. Use stored routines for calculation.
3. Set triggers in MySQL that will handle executions.

## Lesson 1. Creating Compound-Statement Objects

Each stored routine, trigger, or event is an object with a body that must be a single SQL statement. However, these objects often perform complex operations that require several statements. To handle this, you write the statements within a BEGIN ... END block that forms a compound statement. That is, the block is itself a single statement but can contain multiple statements, each terminated by a ; character. The BEGIN ... END block can contain statements such as SELECT or INSERT, but compound statements also allow for conditional statements such as IF or CASE, looping constructs such as WHILE or REPEAT, or otherBEGIN ... END blocks.

Compound-statement syntax provides you with a lot of flexibility, but if you define compound-statement objects within mysql, you'll quickly run into a small problem: statements within a compound statement each must be terminated by a ; character, but mysql itself interprets ; to figure out where each statement ends so that it can send them one at a time to the server to be executed. Consequently, mysql stops reading the compound statement when it sees the first ; character, which is too early. The solution to this problem is to tell mysql to recognize a different statement delimiter. Then mysql will ignore the ; character within the object body. You terminate the object itself with the new delimiter, which mysql recognizes and then sends the entire object definition to the server. You can restore the mysql delimiter to its original value after defining the compound-statement object.

Suppose that you want to define a stored function that calculates and returns the average size in bytes of mail messages listed in the mail table. The function can be defined with a body part consisting of a single SQL statement like this:

```
CREATE FUNCTION avg_mail_size()
RETURNS FLOAT READS SQL DATA
RETURN (SELECT AVG(size) FROM mail);
```

The RETURNS FLOAT clause indicates the type of the function's return value, and READS SQL DATA indicates that the function reads but does not modify data. The body of the function follows those clauses and consists of the single RETURN statement that executes a subquery and returns the value that it produces to the caller. (Every stored function musthave at least one RETURN statement.)

In mysql, you can enter that statement as shown and there is no problem. The definition requires just the single terminator at the end and none internally, so no ambiguity arises. But suppose instead that you want to define the function to take an argument naming a user that is interpreted as follows:

- If the argument is NULL, the function returns the average size for all messages (asbefore).

- If the argument is non-NULL, the function returns the average size for messages sentby that user.

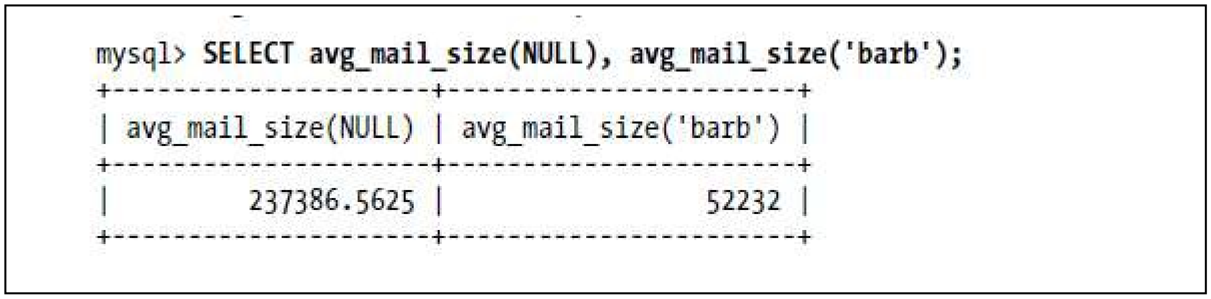
To accomplish this, the routine needs a more complex body that uses a BEGIN ... END block:

```
CREATE FUNCTION avg_mail_size(user VARCHAR(8))
RETURNS FLOAT READS SQL DATA
BEGIN
        IF user IS NULL THEN
                # return average message size over all users
                RETURN (SELECT AVG(size) FROM mail);
        ELSE
                # return average message size for given user
                RETURN (SELECT AVG(size) FROM mail WHERE srcuser = user);
        END IF;
END;
```

If you try to define the function within mysql by entering that definition as is, mysql will improperly interpret the first semicolon in the function body as ending the definition. To handle this, use the delimiter command to change the mysql delimiter to something else temporarily. The following example shows how to do this and then restore the delimiter to its default value:

```
Mysql  > delimiter $$
mysql  > CREATE FUNCTION avg_mail_size (user VARCHAR(8))
        -> RETURNS FLOAT READS SQL DATA
        -> BEGIN
        -> IF user IS NULL THEN
        -> # return average message size over all users
        -> RETURN (SELECT AVG(size) FROM mail);
        -> ELSE
        -> # return average message size for given user
        -> RETURN (SELECT AVG(size) FROM mail WHERE srcuser = user);
        -> END IF;
        -> END;
        -> $$
Query OK, 0 rows affected (0.02 sec)
mysql> delimiter ;
```

After defining the stored function, you can invoke it the same way as built-in functions:


```
mysql> SELECT avg_mail_size(NULL), avg_mail_size('barb');
+---------------------+-----------------------+
| avg_mail_size(NULL) | avg_mail_size('barb') |
+---------------------+-----------------------+
|         237386.5625 |                 52232 |
+---------------------+-----------------------+
```

The same principles apply to defining other objects that use compound statements (stored procedures, triggers, and events (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

## Lesson 2. Using a Stored Function to Encapsulate a Calculation

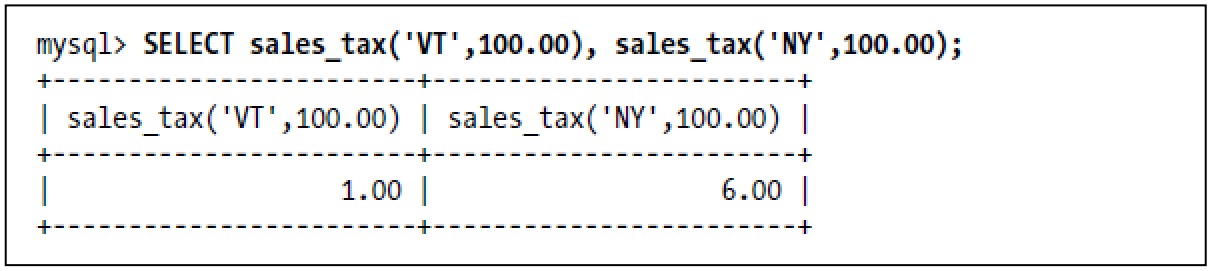
Stored functions enable you to simplify your applications because you can write out the code that produces a calculation result once in the function definition, and then simply invoke the function whenever you need to perform the calculation. Stored functions also enable you to use more complex algorithmic constructs than are available when you write a calculation inline within an expression. This section shows an example that illustrates how stored functions can be useful in these ways. Granted, the example is not actually that complex, but you can apply the same principles used here to write functions that are much more elaborate.

Different states in the U.S. charge different rates for sales tax. If you sell goods to people from different states and must charge tax using the rate appropriate for customer state of residence, tax computation is something you'll need to do for every sale. You can handle this with a table that lists the sales tax rate for each state, and a stored function that calculatesamount of tax given the amount of a sale and a state.

To set up the table, use the sales_tax_rate.sql script in the tables directory of the recipes distribution. The sales_tax_rate table has two columns: state (a two-letter abbreviation), and tax_rate (a DECIMAL value rather than a FLOAT, to preserve accuracy).The stored function, sales_tax( ) can be defined as follows:

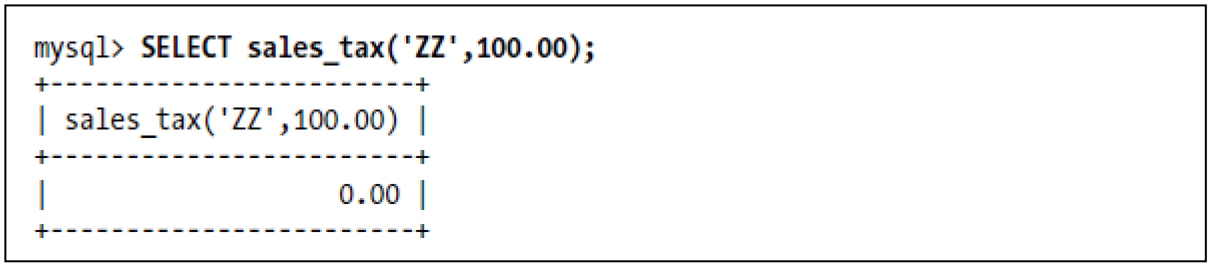
```
CREATE FUNCTION sales_tax(state_param CHAR(2), amount_param DECIMAL(10,2))
RETURNS DECIMAL(10,2) READS SQL DATA
BEGIN
        DECLARE rate_var DECIMAL(3,2);
        DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET rate_var = 0;
        SELECT tax_rate INTO rate_var
                FROM sales_tax_rate WHERE state = state_param;
        RETURN amount_param * rate_var;
END;
```

The function looks up the tax rate for the given state, and returns the tax as the product of the sale amount and the tax rate. Suppose that the tax rates for Vermont and New York are 1 and 9 percent, respectively. Try the function to see whether the tax is computed correctly for a sales amount of $100:



```
mysql> SELECT sales_tax('VT',100.00), sales_tax('NY',100.00);
+------------------------+------------------------+
| sales_tax('VT',100.00) | sales_tax('NY',100.00) |
+------------------------+------------------------+
|                   1.00 |                   6.00 |
+------------------------+------------------------+
```

For a location not listed in the tax rate table, the function should fail to determine a rate and compute a tax of zero:



```
mysql> SELECT sales_tax('ZZ',100.00);
+------------------------+
| sales_tax('ZZ',100.00) |
+------------------------+
|                   0.00 |
+------------------------+
```

Obviously, if you take sales from locations not listed in the table, the function cannot determine the rate for those locations. In this case, the function assumes a tax rate is 0 percent. This is done by means of a CONTINUE handler, which kicks in if a No Data condition (SQLSTATE value 02000) occurs. That is, if there is no row for the given state_param value, the SELECT statement fails to find a sales tax rate. In that case, the CONTINUE handler sets the rate to 0 and continues execution with the next statement after the SELECT. (This handler is an example of the kind of logic that you can use in a stored routine that is not available in inline expressions.) (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

## Lesson 3. Using a Stored Procedure to "Return" Multiple Values

Unlike stored function parameters, which are input values only, a stored procedure parameter can be any of three types:

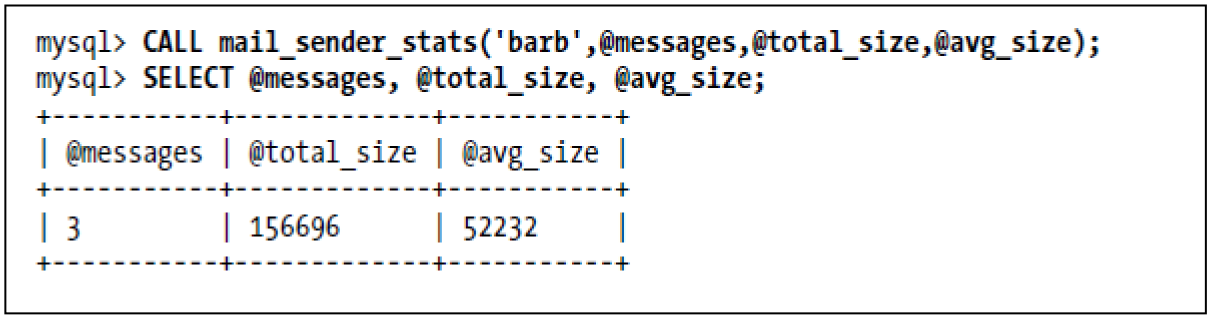
- An IN parameter is for input only. This is the default parameter type if you specify no type.
- An INOUT parameter is used to pass a value in, and it can also be used to pass avalue back out.
- An OUT parameter is used to pass a value out.

This means that if you need to produce multiple values from an operation, you can use INOUT or OUT parameters. The following example illustrates this, using an IN parameter for input, and passing back three values via OUT parameters.

The code from Lesson 1 showed an avg_mail_size( ) function that returns the average mail message size for a given sender. The function returns a single value. If you want additional information, such as the number of messages and total message size, a function will not work. You could write three separate functions, but it's also possible to use a single procedure that retrieves multiple values about a given mail sender. The following procedure, mail_sender_stats( ), runs a query on the mail table to retrieve mail-sending statistics about a given username, which is the input value. The procedure determines how many messages that user sent, and the total and average size of the messages in bytes, which it returns through three OUT parameters:

```
CREATE PROCEDURE mail_sender_stats(IN user VARCHAR(8),
                                   OUT messages INT,
                                   OUT total_size FLOAT,
                                   OUT avg_size FLOAT)
BEGIN
        # Use IFNULL() to return 0 for SUM() and AVG() in case there are
        # no rows for the user (those functions return NULL in that case).
        SELECT COUNT(*), IFNULL(SUM(size),0), IFNULL(AVG(size),0)
        INTO messages, total_size, avg_size
        FROM mail WHERE srcuser = user;
END;
```

To use the procedure, pass a string containing the username, and three user-defined variables to receive the OUT values. After the procedure returns, check the variable values:



```
mysql> CALL mail_sender_stats('barb',@messages,@total_size,@avg_size);
mysql> SELECT @messages, @total_size, @avg_size;
+-----------+-------------+-----------+
| @messages | @total_size | @avg_size |
+-----------+-------------+-----------+
| 3         | 156696      | 52232     |
+-----------+-------------+-----------+
```

## Lesson 4. Using a Trigger to Define Dynamic Default Column Values (Dubois, 2006)

Other than TIMESTAMP columns, which can be initialized to the current date and time, the default value for a column in MySQL must be a constant value. You cannot define a column with a DEFAULT clause that refers to a function call (or other arbitrary expression), and you cannot define one column in terms of the value assigned to another column. That means each of these column definitions is illegal:

```
d               DATE DEFAULT NOW()
i               INT DEFAULT (... some subquery ...)
hashval         CHAR(32) DEFAULT MD5(blob_col)
```

However, you can work around this limitation by setting up a suitable trigger, which enables you to initialize a column however you want. In effect, the trigger enables you to define dynamic (or calculated) default column values.

The appropriate type of trigger for this is BEFORE INSERT, because that enables you to set column values before they are inserted into the table. (An AFTER INSERT trigger can examine column values for a new row, but by the time the trigger activates, it's too late to change the values.)

Suppose that you want to use a table for storing large data values such as PDF or XML documents, images, or sounds, but you also want to be able to look them up quickly later. A TEXT or BLOB data type might be suitable for storing the values, but is not very suitable for finding them. (Comparisons in a lookup operation will be slow for large values.) To work around this problem, use the following strategy:

1. Compute some kind of hash value for each data value and store it in the table along with the data.
2. To look up the row containing a particular data value, compute the hash value for the value and search the table for that hash value. For best performance, make sure that the hash column is indexed.

To implement this strategy, a BEFORE INSERT trigger is helpful because you can have the trigger compute the hash value to be stored in the table for new data values. (If you might change the data value later, you should also set up a BEFORE UPDATE trigger to recompute the hash value.)

The following example assumes that you want to store documents in a table, along with the document author and title. In addition, the table contains a column for storing the hash value computed from the document contents. To generate hash values, the example uses the

First, create a table to hold the document information and the hash values calculated from the document contents. The following table uses a MEDIUMBLOB column to allow storage of documents up to 16 MB in size:

```
CREATE TABLE doc_table
(
        author VARCHAR(100) NOT NULL,
        title VARCHAR(100) NOT NULL,
        document MEDIUMBLOB NOT NULL,
        doc_hash CHAR(32) NOT NULL,
        PRIMARY KEY (doc_hash)
);
```

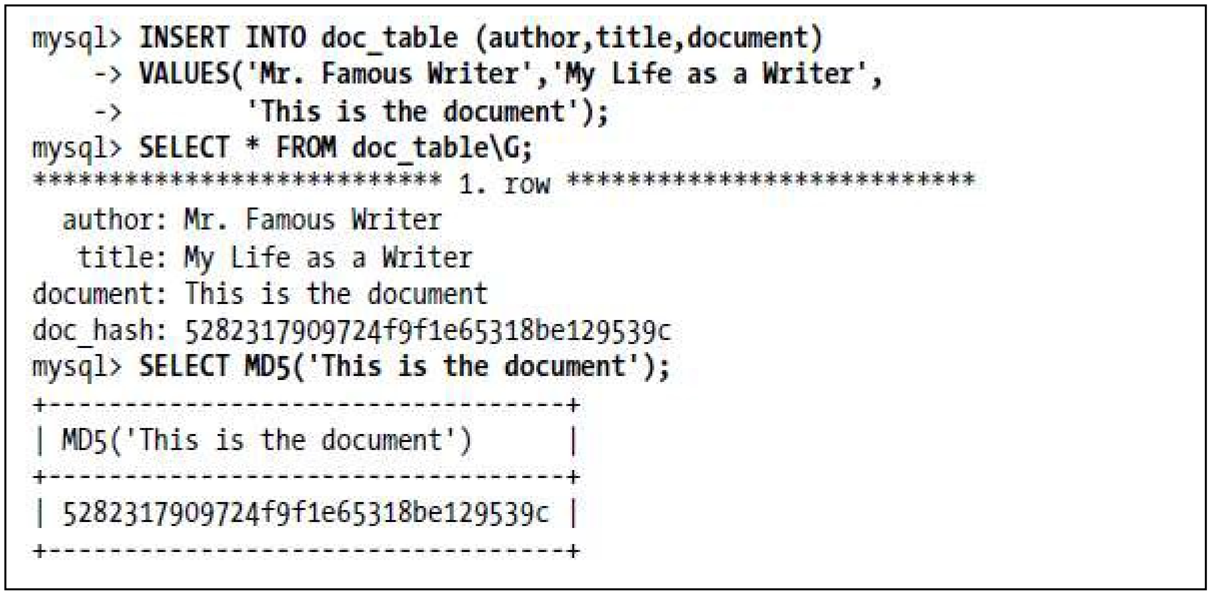
Next, to handle inserts, create a BEFORE INSERT trigger that uses the document to be inserted to calculate the hash value and causes that value to be stored in the table:

```
CREATE TRIGGER bi_doc_table BEFORE INSERT ON doc_table
FOR EACH ROW SET NEW.doc_hash = MD5(NEW.document);
```

This trigger is simple and its body contains only a single SQL statement. For a trigger body that needs to execute multiple statements, use BEGIN ... END compound-statement syntax. In that case, if you use mysql to create the event, you'll need to change the statementdelimiter while you're defining the trigger, as discussed in Lesson 1.

Within the trigger, NEW. col_name refers to the new value to be inserted into the given column. By assigning a value to NEW. col_name within the trigger, you cause the column to have that value in the new row.

Finally, insert a row and check whether the trigger correctly initializes the hash value for the document:

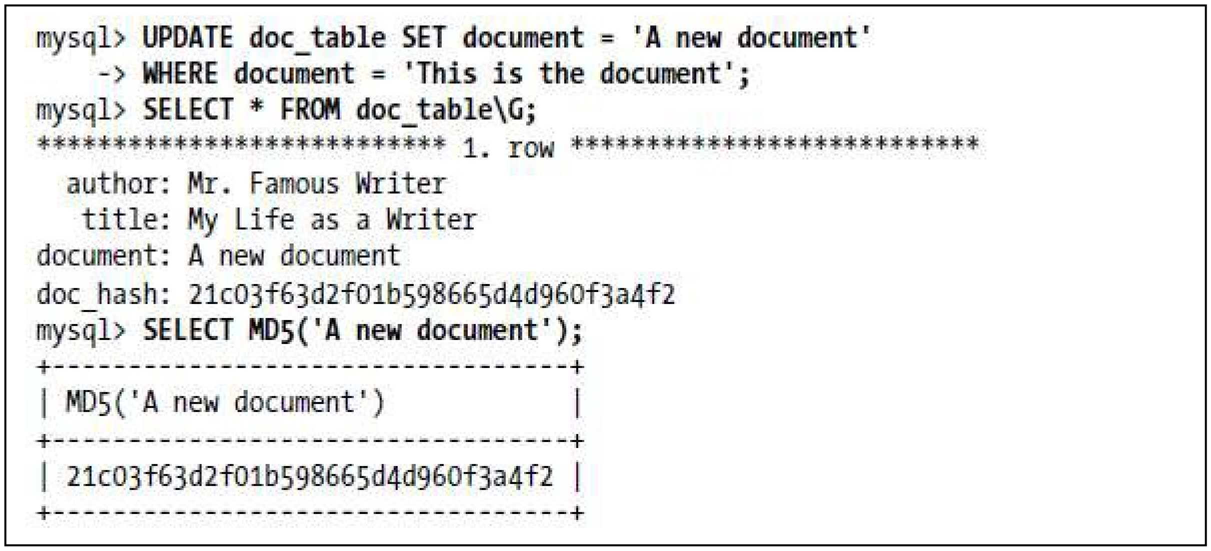

```
mysql> INSERT INTO doc_table (author,title,document)
    -> VALUES('Mr. Famous Writer','My Life as a Writer',
    ->        'This is the document');
mysql> SELECT * FROM doc_table\G;
*************************** 1. row ***************************
  author: Mr. Famous Writer
   title: My Life as a Writer
document: This is the document
doc_hash: 5282317909724f9f1e65318be129539c
mysql> SELECT MD5('This is the document');
+----------------------------------+
| MD5('This is the document')      |
+----------------------------------+
| 5282317909724f9f1e65318be129539c |
+----------------------------------+
```

The first SELECT shows that the doc_hash column was initialized even though the INSERT provided no value for it. The second SELECT shows that the hash value stored in the row by the trigger is correct.

The example thus far demonstrates how a trigger enables you to initialize a row column in a way that goes beyond what is possible with the DEFAULT clause in the column's definition. The same idea applies to updates, and it's a good idea to apply it in the present scenario: when initialization of a column is a function of the value in another column (as is the case for doc_hash), it is dependent on that column. Therefore, you should also update it whenever the column on which it depends is updated. For example, if you update a value in the document column, you should also update the corresponding doc_hash value. This too can be handled by a trigger. Create a BEFORE UPDATE trigger that does the same thing as the INSERT trigger:

```
CREATE TRIGGER bu_doc_table BEFORE UPDATE ON doc_table
FOR EACH ROW SET NEW.doc_hash = MD5(NEW.document);
```

Test the UPDATE trigger by updating the document value and checking whether the hash value is updated properly:



```
mysql> UPDATE doc_table SET document = 'A new document'
    -> WHERE document = 'This is the document';
mysql> SELECT * FROM doc_table\G;
************************** 1. row **************************
  author: Mr. Famous Writer
   title: My Life as a Writer
document: A new document
doc_hash: 21c03f63d2f01b598665d4d960f3a4f2
mysql> SELECT MD5('A new document');
+----------------------------------+
| MD5('A new document')            |
+----------------------------------+
| 21c03f63d2f01b598665d4d960f3a4f2 |
+----------------------------------+
```

## Lesson 5. Simulating TIMESTAMP Properties for Other Date and Time Types (Dubois, 2006)

MySQL supports a TIMESTAMP data type that stores date-and-time values and the conversion of TIMESTAMP values to and from UTC when they are stored and retrieved.

Timestamp special initialization and update properties of the TIMESTAMP data type that enable you to record row creation and modification times automatically. These properties are not available for other temporal types, although there are reasons you might like them to be. For example, if you use separate DATE and TIME columns to store record-modification times, you can index the DATE column to enable efficient datebased lookups. (With TIMESTAMP, you cannot index just the date part of the column.)

One way to simulate TIMESTAMP properties for other temporal data types is to use the following strategy:

- When you create a row, initialize a DATE column to the current date and a TIME column to the current time.
- When you update a row, set the DATE and TIME columns to the new date and time.

However, this strategy requires all applications that use the table to implement the same strategy, and it fails if even one application neglects to do so. To place the burden of remembering to set the columns properly on the MySQL server and not on application writers, use triggers for the table. This is, in fact, a particular application of the general strategy discussed in Lesson 4 that uses triggers to provide calculated values for initializing (or updating) row columns.

The following example shows how to use triggers to simulate TIMESTAMP properties for each of the DATE, TIME, and DATETIME data types. Begin by creating the following table, which has a nontemporal column for storing data and columns for the DATE, TIME, and DATETIME temporal types:

```
CREATE TABLE ts_emulate
(
        data CHAR(10),
        d DATE,
        t TIME,
        dt DATETIME
);
```

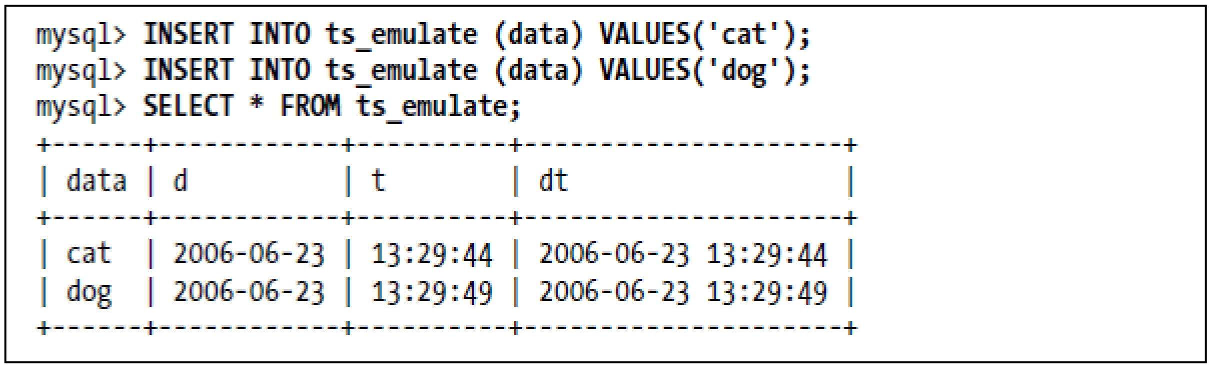
The intent here is that applications will insert or update values in the data column, and MySQL should set the temporal columns appropriately to reflect the time at which modifications occur. To accomplish this, set up triggers that use the current date and time to initialize the temporal columns for new rows, and to update them when existing rows are changed. A BEFORE INSERT trigger handles new row creation by invoking the CURDATE(
), CURTIME( ), and NOW( ) functions to get the current date, time, and date-andtime values and using those values to set the temporal columns:

```
CREATE TRIGGER bi_ts_emulate BEFORE INSERT ON ts_emulate
FOR EACH ROW SET NEW.d = CURDATE(), NEW.t = CURTIME(), NEW.dt = NOW();
```

A BEFORE UPDATE trigger handles updates to the temporal columns when the data column changes value. An IF statement is required here to emulate the TIMESTAMP property that an update occurs only if the values in the row actually change from their current values:

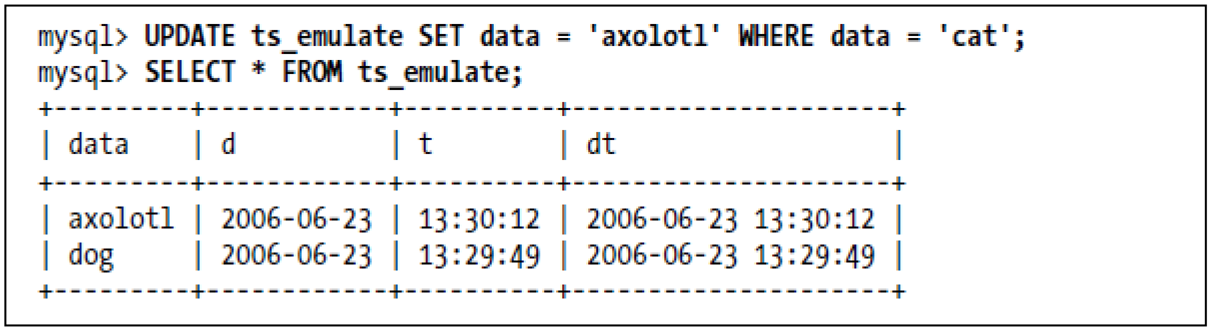
```
CREATE TRIGGER bu_ts_emulate BEFORE UPDATE ON ts_emulate
FOR EACH ROW
BEGIN
        # update temporal columns only if the nontemporal column changes
        IF NEW.data <> OLD.data THEN
                SET NEW.d = CURDATE(), NEW.t = CURTIME(), NEW.dt = NOW();
        END IF;
END;
```

To test the INSERT trigger, create a couple of rows, but supply a value only for the data column. Then verify that MySQL provides the proper default values for the temporal columns:



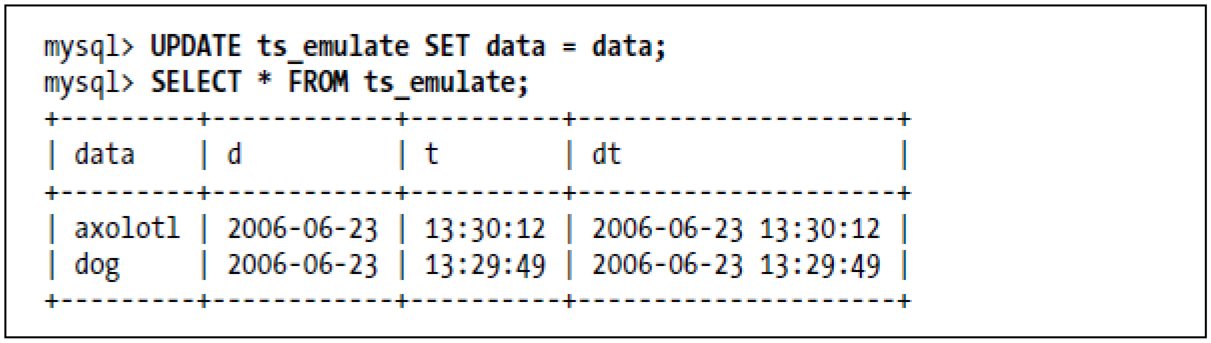
```
mysql> INSERT INTO ts_emulate (data) VALUES('cat');
mysql> INSERT INTO ts_emulate (data) VALUES('dog');
mysql> SELECT * FROM ts_emulate;
+------+------------+----------+---------------------+
| data | d          | t        | dt                  |
+------+------------+----------+---------------------+
| cat  | 2006-06-23 | 13:29:44 | 2006-06-23 13:29:44 |
| dog  | 2006-06-23 | 13:29:49 | 2006-06-23 13:29:49 |
+------+------------+----------+---------------------+
```

Change the data value of one row to verify that the BEFORE UPDATE trigger updates the temporal columns of the changed row:



```
mysql> UPDATE ts_emulate SET data = 'axolotl' WHERE data = 'cat';
mysql> SELECT * FROM ts_emulate;
+---------+------------+----------+---------------------+
| data    | d          | t        | dt                  |
+---------+------------+----------+---------------------+
| axolotl | 2006-06-23 | 13:30:12 | 2006-06-23 13:30:12 |
| dog     | 2006-06-23 | 13:29:49 | 2006-06-23 13:29:49 |
+---------+------------+----------+---------------------+
```

Issue another UPDATE, but this time use one that does not change any data column values. In this case, the BEFORE UPDATE trigger should notice that no value change occurred and leave the temporal columns unchanged:



```
mysql> UPDATE ts_emulate SET data = data;
mysql> SELECT * FROM ts_emulate;
+---------+------------+----------+---------------------+
| data    | d          | t        | dt                  |
+---------+------------+----------+---------------------+
| axolotl | 2006-06-23 | 13:30:12 | 2006-06-23 13:30:12 |
| dog     | 2006-06-23 | 13:29:49 | 2006-06-23 13:29:49 |
+---------+------------+----------+---------------------+
```

The preceding example shows how to simulate the auto-initialization and auto-update properties offered by TIMESTAMP columns. If you want only one of those properties and not the other, create only one trigger and omit the other (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## Lesson 6. Using a Trigger to Log Changes to a Table

Suppose that you conduct online auctions, and that you maintain information about each currently active auction in a table that looks like this:

```
CREATE TABLE auction
(
        id INT UNSIGNED NOT NULL AUTO_INCREMENT,
        ts TIMESTAMP,
        item VARCHAR(30) NOT NULL,
        bid DECIMAL(10,2) NOT NULL,
        PRIMARY KEY (id)
);
```

The auction table contains information about the currently active auctions (items being bid on and the current bid for each auction). When an auction begins, you enter a row into the table. Its bid column gets updated for each new bid on the item. When the auction ends, the bid value is the final price and the row is removed from the table. As the auction proceeds, the ts column is updated to reflect the time of the most recent bid.

If you also want to maintain a journal that shows all changes to auctions as they progress from creation to removal, you can modify the auction table to allow multiple records per item and add a status column to show what kind of action each row represents. Or you could leave the auction table unchanged and set up another table that serves to record a history of changes to the auctions. This second strategy can be implemented with triggers.

To maintain a history of how each auction progresses, use an auction_log table with the following columns:

```
CREATE TABLE auction_log
(
        action ENUM('create','update','delete'),
        id INT UNSIGNED NOT NULL,
        ts TIMESTAMP,
        item VARCHAR(30) NOT NULL,
        bid DECIMAL(10,2) NOT NULL,
        INDEX (id)
);
```

The auction_log table differs from the auction table in two ways:

- It contains an action column to indicate for each row what kind of change was made.
- The id column has a nonunique index (rather than a primary key, which requires unique values). This allows multiple rows per id value because a given auction can generate many rows in the log table.

To ensure that changes to the auction table are logged to the auction_log table, create a set of triggers. The triggers should write information to the auction_log table as follows:

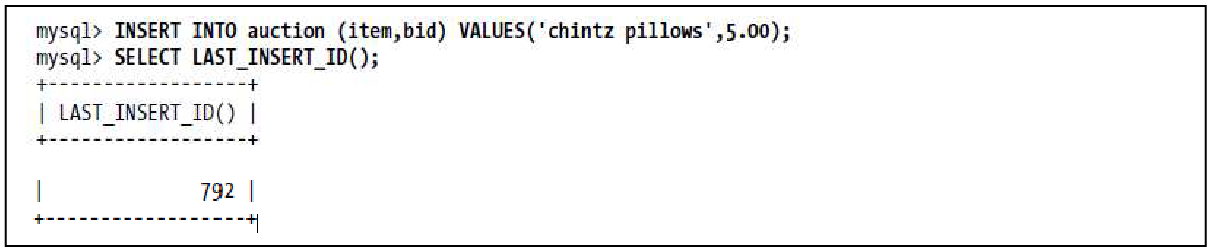
- For inserts, log a row-creation operation showing the values in the new row.
- For updates, log a row-update operation showing the new values in the updated row.
- For deletes, log a row-removal operation showing the values in the deleted row.

For this application, AFTER triggers are used, because they will activate only after successful changes to the auction table. (BEFORE triggers might activate even if the rowchange operation fails for some reason.) The trigger definitions look like this:

```
CREATE TRIGGER ai_auction AFTER INSERT ON auction
FOR EACH ROW
BEGIN
            INSERT INTO auction_log (action,id,ts,item,bid)
            VALUES('create',NEW.id,NOW(),NEW.item,NEW.bid);
END;

CREATE TRIGGER au_auction AFTER UPDATE ON auction
FOR EACH ROW
BEGIN
            INSERT INTO auction_log (action,id,ts,item,bid)
            VALUES('update',NEW.id,NOW(),NEW.item,NEW.bid);
END;

CREATE TRIGGER ad_auction AFTER DELETE ON auction
FOR EACH ROW
BEGIN
            INSERT INTO auction_log (action,id,ts,item,bid)
            VALUES('delete',OLD.id,OLD.ts,OLD.item,OLD.bid);
END;
```

The INSERT and UPDATE triggers use NEW. col_name to access the new values being stored in rows. The DELETE trigger uses OLD. col_name to access the existing values from the deleted row. The INSERT and UPDATE triggers use NOW( ) to get the row- modification times; the ts column is initialized automatically to the current date and time, butNEW.ts will not contain that value.

Suppose that an auction is created with an initial bid of five dollars:



```
mysql> INSERT INTO auction (item,bid) VALUES('chintz pillows',5.00);
mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+

|              792 |
+------------------+
```

The SELECT statement fetches the auction ID value to use for subsequent actions on the auction. Then the item receives three more bids before the auction ends and is removed:

```
mysql> UPDATE auction SET bid = 7.50 WHERE id = 792;
                    ... time passes ...
mysql> UPDATE auction SET bid = 9.00 WHERE id = 792;
                    ... time passes ...
mysql> UPDATE auction SET bid = 10.00 WHERE id = 792;
                    ... time passes ...
mysql> DELETE FROM auction WHERE id = 792;
```

At this point, no trace of the auction remains in the auction table, but if you query the auction_log table, you can obtain a complete history of what occurred:

```
mysql> SELECT * FROM auction_log WHERE id = 792 ORDER BY ts;
+--------+-----+---------------------+----------------+-------+
| action | id  | ts                  | item           | bid   |
+--------+-----+---------------------+----------------+-------+
| create | 792 | 2006-06-22 21:24:14 | chintz pillows |  5.00 |
| update | 792 | 2006-06-22 21:27:37 | chintz pillows |  7.50 |
| update | 792 | 2006-06-22 21:39:46 | chintz pillows |  9.00 |
| update | 792 | 2006-06-22 21:55:11 | chintz pillows | 10.00 |
| delete | 792 | 2006-06-22 22:01:54 | chintz pillows | 10.00 |
+--------+-----+---------------------+----------------+-------+
```

With the strategy just outlined, the auction table remains relatively small, but we can always find information about auction histories as necessary by looking in the auction_log table.

## Lesson 7. Using Events to Schedule Database Actions

(Dubois, 2006)

As of MySQL 5.1, one of the capabilities available to you is an event scheduler that enables you to set up database operations that run at times that you define. This section describes what you must do to use events, beginning with a simple event that writes a row to a table at regular intervals. Why bother creating such an event? One reason is that the rows serve as a log of continuous server operation, similar to the MARK line that some Unix syslogdservers write to the system log periodically so that you know they're alive.

Begin with a table to hold the mark records. It contains a TIMESTAMP column (which MySQL will initialize automatically) and a column to store a message:

```
mysql> CREATE TABLE mark_log (ts TIMESTAMP, message VARCHAR(100));
```

Our logging event will write a string to a new row. To set it up, use a CREATE EVENTstatement:

```
mysql> CREATE EVENT mark_insert
        -> ON SCHEDULE EVERY 5 MINUTE
        -> DO INSERT INTO mark_log (message) VALUES('-- MARK --');
```

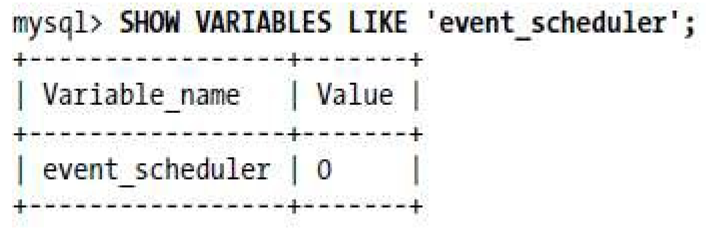
The mark_insert event causes the message '-- MARK --' to be logged to the mark_logtable every five minutes. Use a different interval for more or less frequent logging.

This event is simple and its body contains only a single SQL statement. For an event body that needs to execute multiple statements, use BEGIN ... END compound-statement syntax. In that case, if you use mysql to create the event, you need to change the statement delimiter while you're defining the event, as discussed in Lesson 1.

At this point, you should wait a few minutes and then select the contents of the mark_log table to verify that new rows are being written on schedule. However, if this is the first event that you've set up, you might find that the table remains empty no matter how long you wait:

```
mysql> SELECT * FROM mark_log;
Empty set (0.00 sec)
```

If that's the case, very likely the event scheduler isn't running (which is its default state until you enable it). Check the scheduler status by examining the value of the event_schedulersystem variable:



```
mysql> SHOW VARIABLES LIKE 'event_scheduler';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| event_scheduler | 0     |
+-----------------+-------+
```

To enable the scheduler interactively if it is not running, execute the following statement (which requires the SUPER privilege):

```
mysql> SET GLOBAL event_scheduler = 1;
```

That statement enables the scheduler, but only until the server shuts down. To make sure that the scheduler runs each time the server starts, set the system variable to 1 in your my.cnf option file:

```
[mysqld]
event_scheduler=1
```

When the event scheduler is enabled, the mark_insert event eventually will create many rows in the table. There are several ways that you can affect event execution to preventthe table from growing forever:

- Drop the event:
    - mysql> DROP EVENT mark_insert;

  This is the simplest way to stop an event from occurring. But if you want it to resume later, you must re-create it.

- Suspend execution for the event:
    - mysql> ALTER EVENT mark_insert DISABLE;

  Disabling an event leaves it in place but causes it not to run until you reactivate it:

    - mysql> ALTER EVENT mark_insert ENABLE;

- Let the event continue to run, but set up another event that “expires” old mark_log rows. This second event need not run so frequently (perhaps once a day). Its body should contain a DELETE statement that removes rows older than a given threshold. The following definition creates an event that deletes rows that are more than two daysold:

  -
    ```
    mysql> CREATE EVENT mark_expire
              -> ON SCHEDULE EVERY 1 DAY
              -> DO DELETE FROM mark_log WHERE ts < NOW() - INTERVAL 2 DAY;
    ```

  If you adopt this strategy, you have cooperating events, such that one event adds rows to the mark_log table and the other removes them. They act together to maintain a log that contains recent records but does not become too large.

To check on event activity, look in the server’s error log, where it records information about which events it executes and when.

## Assessment Task

Based from your understanding, write a brief discussion in the use of stored Routines,Triggers, and Events. (2-3 paragraph)

***Note: Your answer will be tested with Plagiarism checker.***

## Summary

- A stored function performs a calculation and returns a value that can be used in expressions just like a built-in function such as RAND( ), NOW( ), or LEFT( ).

- A trigger is an object that is defined to activate when a table is modified. Triggers are available for INSERT, UPDATE, and DELETE statements.
- Events An event is an object that executes SQL statements at a scheduled time or times. You can think of an event as something like a Unix cron job, but that runs within MySQL.

## References

Kruckenberg, M., & Pipes, J. (2005). Pro MySQL. In Pro MySQL.https://doi.org/10.1007/978-1-4302-0048-2

Ramakrishhan, Raghu and Gehrke,Johannes (2018) ® Database Management Systems. McGraw Hill

(Ramakrishhan, Raghu and Gehrke,Johannes,2018)

Source :SQL for beginners: A guide to study SQL programming and database management systems

Database management systemsGupta, G. K.,2020

# MODULE 7
# CURSORS

## Introduction

Cursors, like stored procedures and functions, are new to MySQL in version 5.0. Cursors are a welcome addition for people who are coming from cursor-capable databases or have been using MySQL but grumbling about not being able to use cursors. Most major databases provide cursor functionality, including DB2, Oracle, SQL Server, Sybase, and PostgreSQL. With 5.0, MySQL joins the ranks of other cursor-capable database systems (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## Learning Outcomes

At the end of this module, students should be able to:

1. Use database cursors.
2. Explain the use of cursors
3. Differentiate cursor from handler.

## Lesson 1. Database Cursors

You might have places where using single pieces of data meets your needs, but you're probably left thinking that unless a routine can work with sets of data, stored procedures or functions aren't of much use to you. For example, you might have a stored procedure that needs to be capable of processing a large set of records, inserting data into a set of tables based on the values in each record. Doing all of this within a stored procedure makes a lot of sense, but you need to be able to issue a SELECT statement, retrieve the rows, and loop through them one at a time to determine how each record should be handled.

In database terms, a cursor is a pointer to a record in a set of results in the database. Using a database cursor allows you to issue a query, but rather than getting back the set of query records, you get back a pointer to the data that allows you to interact with the set of records. The cursor provides a mechanism to use the data, allowing you to issue commands to read information, move to another record, make a change to the data, and so on. Once the cursor has been created, it remains actively pointed at the data until it is closed or the connection to the database is closed.

A popular use of cursors is to issue a query that requests a cursor in return, and then have code control the cursor to iterate through the set of records, performing logic based on the information in each row.

As with stored procedures, stored functions, and other technologies, there is a debate as to where cursors are appropriate. Some suggest that cursors should be a last resort. They say that, in most cases, it's better to return the entire resultset to the client to work with than to tie up the database keeping track of a cursor over a period of time. Others suggest that cursors are a preferred method for interacting with large sets of data. They say that using cursors allows for more immediate access to the data and reduces the load on the database, because cursors can return the data to the client incrementally, and only as needed by the client. Whether cursors are appropriate for your application is a question you must answer when looking at the needs of your users and the stewards of the data. Before you decide to invest time and energy into using cursors, you should carefully weigh the implications of moving data processing for multiple-row data sets out of your application and into the database (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

**Server and Client Cursors**

The two major types of cursors are server-side cursors and client-side cursors. Server- side cursors let you open a cursor in code that is run inside the database. You are not able to send the cursor, or a pointer to control the cursor, to an external client for interaction with the database. Server-side cursors are opened, used, and closed from within a routine inside the database. The cursor is opened and closed without any interaction with an external client, other than calling the procedure that may use a cursor internally.

from a client outside the database (for example, in your application) and have the database return the cursor to the external client for control. Where client-side cursors are in use, you will see the client or application make a query to the database that asks for a cursor, instead of the record set, in the return. The client gets the cursor from the database, and then uses logic built into the application to control the cursor's movement, retrieval, and modification of data. Once the application has finished, it is expected to close the cursor.

MySQL offers only server-side cursors.

**Cursor Movement**

Depending on the database, control over the movement of the cursor varies. In its simplest form, a cursor moves forward one record at a time and gives you no control over the direction or spacing of the movement.

More sophisticated implementations will allow you to move the cursor both forward and backward. Beyond the ability to move backward is the capability to skip to certain records, based on either a record number or a position relative to where the cursor is currently positioned. It's also common to see a command to move the cursor back to the first or forward to the last record. MySQL cursors are of the forward-only type.

**Data Behind the Cursor**

You might wonder exactly where the cursor resides and what data it's using when you're scrolling around. The SQL standard specifies that cursors can be either insensitive or asensitive.

An insensitive cursor is one that points at a temporary copy of the data. Any changes in the data while the cursor is open are hidden from the cursor, because the cursor is looking at a snapshot of the data taken at the time the cursor was requested. The snapshot of the data sticks around until the cursor is closed.

An asensitive cursor points at the real data, not a cached or temporary copy. A cursor that points at the actual data becomes available faster than an insensitive cursor, because the data doesn't need to be copied to a temporary place. However, when using a cursor that points at the actual data, changes in the underlying data from another connection may affectthe data being used by the cursor. MySQL cursors are asensitive.

**Read and Write Cursors**

The most common type of cursor returned from a database is a cursor used for reading data. However, some cursors can make changes to the record where the cursor is currently positioned. The ability to write via a database cursor opens up a lot of additional options for your cursor use. With a write cursor, you could run a query, perform some logic on the fields in the query, and make updates to the fields in the record based on the logic. To do this same kind of thing from the client would take getting the entire resultset and then performing numerous updates on each row. MySQL cursors are read-only. (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

## Lesson 2. Cursors in MySQL

MySQL comes with cursor functionality in version 5.0 and later. As you would expect, MySQLhas implemented a small core of cursor functionality that can accomplish a lot.

MySQL's cursor implementation can be summarized in one statement: MySQL cursors are read-only, server-side, forward-moving, and asensitive. As our earlier discussion indicated, because they are server-side cursors, they can be used only within stored routines in the database. You cannot request a cursor from a client, and the database is not capable of returning a cursor for external control.

Being ready-only, MySQL cursors are limited to the use of the cursor to retrieve information from the database, not to make changes. If you want to make changes to the data retrieved through a cursor, you'll need to issue a separate UPDATE statement. MySQL cursors can move only forward, one record at a time.

The FETCH statement is used to move the cursor forward one record. If you need to move forward more than one record— for example, because you want to process every fifth record—you could use IF statements with a variable to perform your logic every so many FETCH statements.

MySQL supports asensitive, but not insensitive, cursors. This means that when you open a cursor to read information from a table, the actual table data is used when grabbing information, as opposed to a cached copy or temporary table to isolate the cursor's interaction with the data from other client interaction. With cursors in MySQL being asensitive, you aren't required to wait until the data is loaded from the main table storage to a temporary location to start fetching the data. This also means that you might find that data changes through other connections to the database at the time you are using your cursor. These changes may show up as you move through the records retrieved by your cursor declaration. (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

## Lesson 3. Creating Cursors

Let's start with a simple example so you can get some idea of what a function with a cursor looks like. For this example, suppose we have been asked to build an easy way for employees to get a list of the cities where we have order-processing facilities. Since we took great care to normalize our data, all cities are stored in a central table. The employees could just SELECT from the city table, but the vertical list of city names output from the SELECT isn't acceptable.

The employees use this information to dynamically generate documents that must put the city names in a sentence

We could write a tool that goes to the database, gets the list, and formats the results in a string to be used in a paragraph, or we could put this into a function and let the employees call the function. Listing 7.1 shows how we might build the city_list() function.Ramakrishhan,RaghuandGehrke,Johannes,2018)

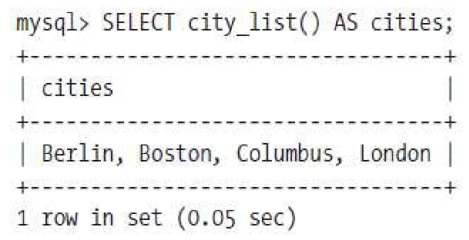
**Listing 7.1. city_list() Function**

```
DELIMITER //
CREATE FUNCTION city_list() RETURNS VARCHAR(255)
BEGIN
        DECLARE finished INTEGER DEFAULT 0;
        DECLARE city_name VARCHAR(50) DEFAULT "";
        DECLARE list VARCHAR(255) DEFAULT "";
        DECLARE city_cur CURSOR FOR SELECT name FROM city ORDER BY name;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

        OPEN city_cur;

        get_city: LOOP
                FETCH city_cur INTO city_name;
                IF finished THEN
                        LEAVE get_city;
                END IF;
                SET list = CONCAT(list,", ",city_name);
        END LOOP get_city;
        CLOSE city_cur;
        RETURN SUBSTR(list,3);
END
//
DELIMITER ;
```

When entering multiple statement blocks into MySQL, you need to first change the default delimiter to something other than a semicolon (;), so MySQL will allow you to enter a ; without having the client process the input. Listing 7.1 begins by using the delimiter statement: DELIMITER //, which changes the delimiter to //. When you're ready to have your procedure created, type //, and the client will process your entire procedure. When you're finished working on your procedures, change the delimiter back to the standard semicolon with DELIMITER ;, as you can see at the end of Listing 7.1.

Before we look at the syntax for the statements related to cursors, let's step through what's happening in the city_list() function. It starts with a number of variable declarations, a cursor declaration to get city names from the city table, and a handler declaration to help you determine when you've reached the end of the results. It then opens the cursor and iterates through the results, fetching each record, checking if the fetch failed, and adding the name of the current city to our string. Finally, it returns the string to the caller. Using this function is simple, as shown in Listing 7.2.

**Listing 7.2. Using the city_list() Function**



```
mysql> SELECT city_list() AS cities;
+---------------------------------+
| cities                          |
+---------------------------------+
| Berlin, Boston, Columbus, London |
+---------------------------------+
1 row in set (0.05 sec)
```

The employees who need to have a string list of all the cities in the system can now use this function where required. It will always get the list of current city names in the city table. The statements for using cursors are intertwined with statements to build the overall routine, but appear in the order DECLARE, OPEN, FETCH, and CLOSE as you build a procedure or function that uses a cursor(Ramakrishhan, Raghu and Gehrke,Johannes,2018)

## Lesson 4. DECLARE Statements

When building a stored procedure or function that requires a cursor, a few DECLARE statements are needed. You may recall from our coverage of stored procedures in previous modules that declarations must be ordered variables first, cursors second, and handlers last.

### Variables

When using a cursor, you need to declare at least two variables. One is the variable that will be used to indicate that the cursor has reached the end of the record set. In Listing 11-1, we used the finished variable for this purpose:

```
DECLARE finished INTEGER DEFAULT 0;
```

The finished variable was initialized to 0. During the loop over the record, this variable is checked with an IF statement to determine if the last record has been reached. We'll look at the loop shortly, when we discuss the HANDLER statement.

Beyond the variable for exiting the loop, you also need to declare a variable for each field that you will FETCH from the cursor. In this example, we're getting only one field from each row, so we declare only one variable to store the field value being pulled from the row of data:

```
DECLARE city_name VARCHAR(50) DEFAULT "";
```

When you're iterating over the records, the FETCH statement gets the field value from the cursor and assigns the value to the variable. We'll cover fetching data from the records inthe upcoming section about the FETCH statement.

Each time the loop iterates, city_name will be set to the value of name for the current row and be available for use in whatever logic you're performing in the loop. In this case, we're building a string. We also have declared a list variable that is created to store the joined city names. This variable will eventually be used for the return to the client.

**Cursor**

Once we've declared the variables for the procedure, we can declare the cursor itself. The statement to create the cursor is shown here:

```
DECLARE <cursor name> CURSOR FOR <SELECT statement>;
```

This statement simply defines the cursor, but does not actually process the statement or create the pointer to the data. The DECLARE statement from Listing 11-1 shows how this looks in practice:

```
DECLARE city_cur CURSOR FOR SELECT name FROM city ORDER BY name;
```

Here, we create a cursor named city_cur, which is defined as a pointer to the data froma SELECT statement that will retrieve all the names from the city table.

## Handler

A HANDLER statement is required to detect and handle when the cursor cannot find any more records. Each time the FETCH statement is processed, it attempts to get the next row of data. When it has reached the end of the set of results, it will not be able to find another set of data. At that time, a condition will be raised, the handler will be activated, and the handler will set up for the iteration over the records to exit.

In the example in Listing 7.1, we declared a finished variable, initially set to 0. We then created a handler that says, "When FETCH has raised the condition that it couldn't find a record to read, set the finished variable to 1," as follows:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
```

Each time the loop iterates, it attempts to FETCH, and then immediately checks the finished variable to see if it was changed by the condition being raised. If the finished variable wasn't changed by an activation of the HANDLER statement, the loop continues. This logic isrepeated for each iteration through the loop:

```
get_city: LOOP
        FETCH city_cur INTO city_name;
        IF finished THEN
        LEAVE get_city;
        END IF;
        SET list = CONCAT(list,", ",city_name);
END LOOP get_city;
```

If the FETCH can't get a row, a NOT FOUND condition is raised. This condition is handled by the handler, which sets finished to 1. The IF finished THEN statement will then be satisfied, and the loop will exit on the LEAVE statement. A loop label, get_city in this example,is required to use the LEAVE statement.

***Note:***

You may be tempted to use the REPEAT statement for your iterations, but be careful. The REPEAT mechanism is problematic because it doesn't allow you to check to see if the FETCH caused the reset of the finished variable until the bottom of the loop. If you don’t check if the loop should exit immediately after the FETCH, you will execute statements, even though the end of the resultset has been reached.

**OPEN Statement**

After you have declared the necessary variables and the cursor itself, you can activatethe cursor by using the OPEN statement:

```
OPEN <cursor name>;
```

For our example, where the cursor was declared as city_cur, the OPEN statement looks like:

```
OPEN city_cur;
```

When you issue this statement, the SELECT statement in the cursor declaration is processed, and the cursor is pointed at the first record in the statement, ready for a FETCHto happen.

***NOTE:***

A cursor declaration can be opened multiple times within a stored procedure. If the cursor has been closed, the OPEN statement can be used to run the SQL statement again and give you a pointer to the data. This might be helpful if you have a set of records that needs to be usedseveral times in a procedure or function.

**FETCH Statement**

The FETCH statement gets the data from the record, assigns it to a variable, and tells the cursor to move to the next record. The FETCH statement requires a cursor name and a variable:

```
FETCH <cursor name> INTO <variable name>[, <variable name>, …];
```

The example in Listing 7.1 fetches the record from the city_cur cursor and puts that single value into the city_name variable:

```
FETCH city_cur INTO city_name;
```

If your SELECT statement in DECLARE…CURSOR contains more than one column, you'll be required to provide a comma-delimited list of variables to use when assigning the value of each field to a variable. For example, if the SELECT statement in Listing 7.1 had also specified the city_id, we would have needed to declare a variable to hold the city_id when it was fetched.

**CLOSE Statement**

When you are finished with the cursor, you should close it. The CLOSE statement to accomplish this is simple:

```
CLOSE <cursor name>;
```

In Listing 7.1, we close the city_cur cursor as follows:

```
CLOSE city_cur;
```

Closing the cursor removes the pointer from the data. If you don't issue the CLOSE statement, the cursor will remain open until your connection to the database is closed.

**Note:**

You can have multiple cursors within a single stored procedure, each with its own DECLARE statement. You can have them opened at the same time and FETCH from them within the same loop, or in separate iterations at different points in your procedure or function. Be aware that the NOT FOUND condition will be met when any of the FETCH statements finds that it is at the end of the record set. There is no way to assign a condition to a specific cursor.

## Lesson 5. Using Cursors

We started the chapter with a scenario where you needed to process multiple records and move them to a set of other tables in your system, with the appropriate table for a row being decided by the data in the row. Now, let's see how you might solve this type of problem with a stored procedure and cursor.

Sticking with the online store example we've been using in previous chapters, suppose we have a login table that keeps track of when a customer logs in to our system. We use this information to keep a customer login history for statistics, as well as to record login information in case a security issue arises. We also use this table to show users their login history for the past week. Because we have user counts in the hundreds of thousands at any given point, and users log in to the system many times throughout each day, the table that tracks their access grows quickly, as expected. Within just a few months, the login process, which checks the last login and inserts a new record, and the page that displays the recent login informationboth show signs of problems with scalability.

Since we really want only the logins from the past week, we don't need all the information to stay in the login table. To reduce the size of our login table, we decide to create a login_archive table and push the older information in the login table into this archive. From our customer support group and reps, we learn that the data would be even more useful if it were separated using the region of the user and placed in tables replicated out to the regional offices during the archive process. To do this, we create an archive table for each of three main regions: login_archive_south, login_archive_northeast, and login_archive_ northwest. However, we still need a way to go through all of the data and move it to the right place.

This scenario is perfect for a stored procedure that uses a cursor to go through each line of the data and move it into different tables. Since we don't actually need to see any of the data in the client during this process, using a stored procedure will give us better performance, because data does not need to flow back and forth between the client and the database. The interaction with the data and the data itself stays inside the database until the process is complete. Listing 7.3 shows the login_archive() procedure built to accomplish this task (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

**Listing 7.3. login_archive() Procedure**

```
DELIMITER //
CREATE PROCEDURE login_archive()
BEGIN
	DECLARE finished INTEGER DEFAULT 0;
	DECLARE cust_id INTEGER;
	DECLARE log_id INTEGER;
	DECLARE time DATETIME;
	DECLARE moved_count INTEGER DEFAULT 0;
	DECLARE customer_region INTEGER;

	DECLARE login_curs CURSOR FOR SELECT l.customer_id, l.login_time,
		c.region, l.login_id
		FROM login l, customer c
		WHERE l.customer_id = c.customer_id
		AND to_days(l.login_time) < to_days(now())-7;

	DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

	OPEN login_curs;

	move_login: LOOP

	FETCH login_curs INTO cust_id, time, customer_region, log_id;

	IF finished THEN
		LEAVE move_login;
	END IF;

	IF customer_region = 1 THEN
		INSERT INTO login_northeast SET customer_id = cust_id,
		login_time = time;
	ELSEIF customer_region = 2 THEN
		INSERT INTO login_northwest SET customer_id = cust_id,
		login_time = time;
	ELSE
		INSERT INTO login_south SET customer_id = cust_id,
		login_time = time;
	END IF;
```

**Listing 7.3. login_archive() Procedure Continuation**

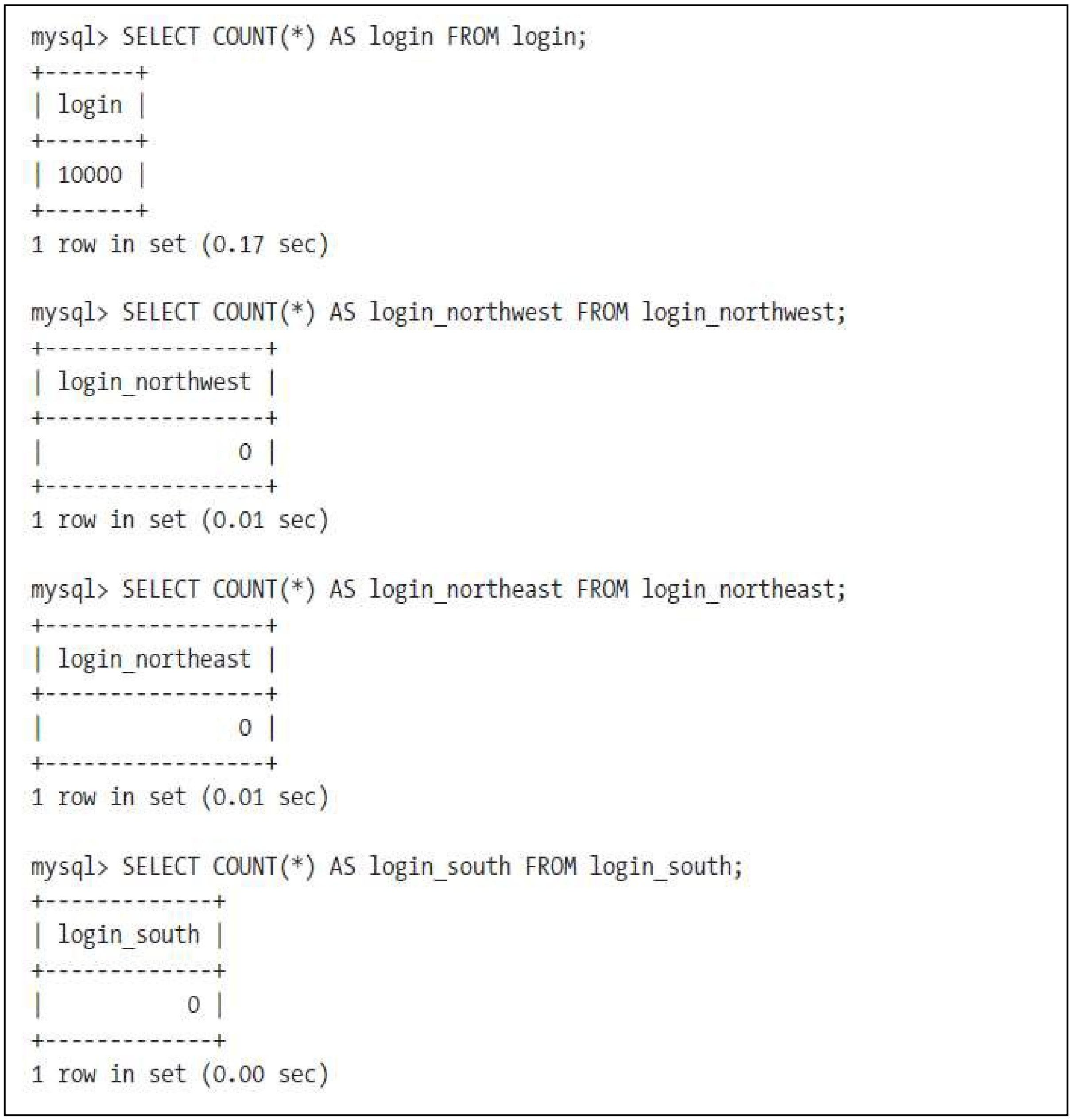
```
        DELETE from login where login_id = log_id;

        SET moved_count = moved_count + 1;

        END LOOP move_login;

        CLOSE login_curs;

        SELECT moved_count as 'records archived';
END
//
DELIMITER ;
```

This stored procedure starts with the expected declaration of the finished variable and other variables for storing fetched data, as well as the cursor and handler declaration. We also create the moved_count variable for keeping track of how many records are moved. The SQL statement that defines the cursor is a bit more complex than our first example, but nothing out of the ordinary. It joins two tables and includes a WHERE clause to limit the results to recordsthat are older than seven days.

We then open the cursor and use the LOOP statement to go through each record. Notice that the FETCH statement assigns values to multiple variables, correlated with the number of columns in the SELECT statement that defined city_cur. After the FETCH, we first check to see if we’ve reached the end of the records, exiting if we have. If we’re not ready to exit, we use the values in the record to determine which region the customer belongs in, and INSERT the record into the appropriate table. Finally, in the loop, we DELETE the record that was just moved so it isn’t in the login table (one of the main purposes for the archive process)and increment the moved_count variable.

Let’s try this to make sure that the procedure is moving the data appropriately. A look at the record counts shows that there’s a lot of information in the login table, and none in the region-specific tables. Listing 7.4 shows the output from four queries to get record counts. (Ramakrishhan, Raghu and Gehrke,Johannes,2018)

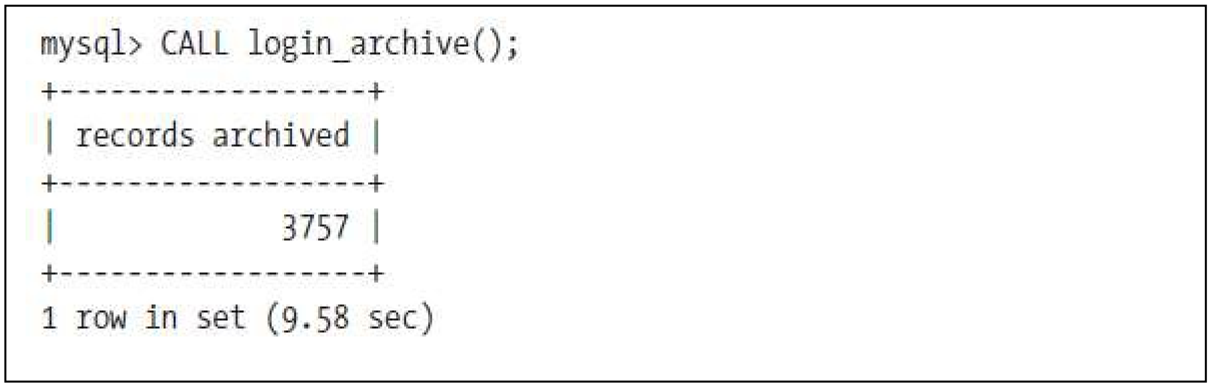
**Listing 7.4. Record Counts Before Calling archive_login()**



```
mysql> SELECT COUNT(*) AS login FROM login;
+-------+
| login |
+-------+
| 10000 |
+-------+
1 row in set (0.17 sec)

mysql> SELECT COUNT(*) AS login_northwest FROM login_northwest;
+-----------------+
| login_northwest |
+-----------------+
|               0 |
+-----------------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) AS login_northeast FROM login_northeast;
+-----------------+
| login_northeast |
+-----------------+
|               0 |
+-----------------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) AS login_south FROM login_south;
+-------------+
| login_south |
+-------------+
|           0 |
+-------------+
1 row in set (0.00 sec)
```

Now, let’s see what happens when we execute the stored procedure, with its internal cursor mechanism, using the CALL statement, as shown in Listing 7.5.

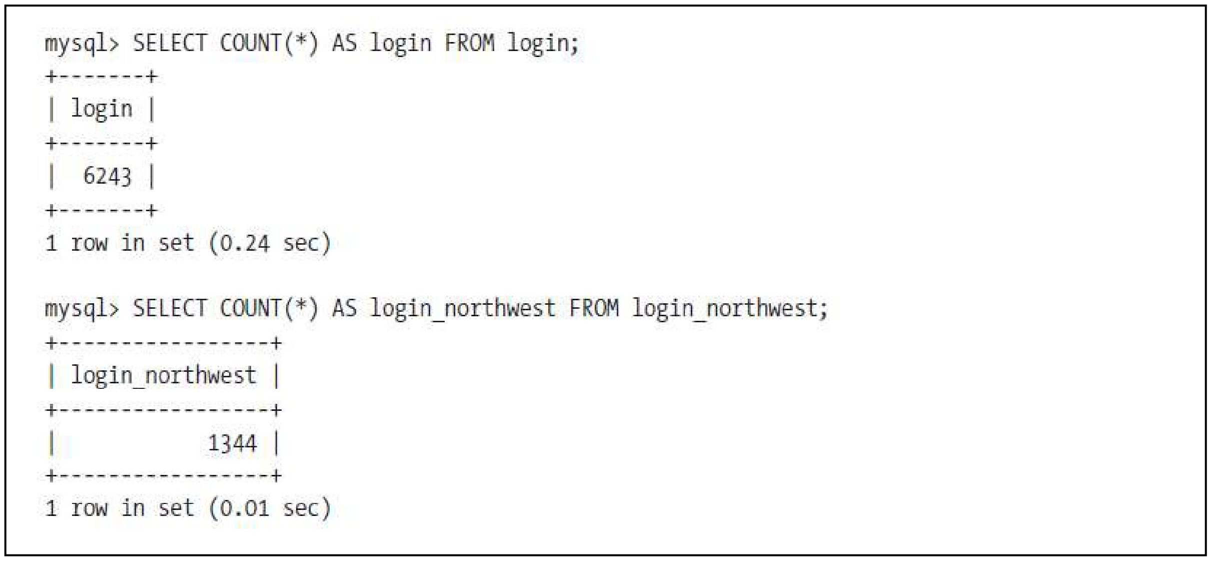
**Listing 7.5. Running login_archive()**



```
mysql> CALL login_archive();
+------------------+
| records archived |
+------------------+
|             3757 |
+------------------+
1 row in set (9.58 sec)
```

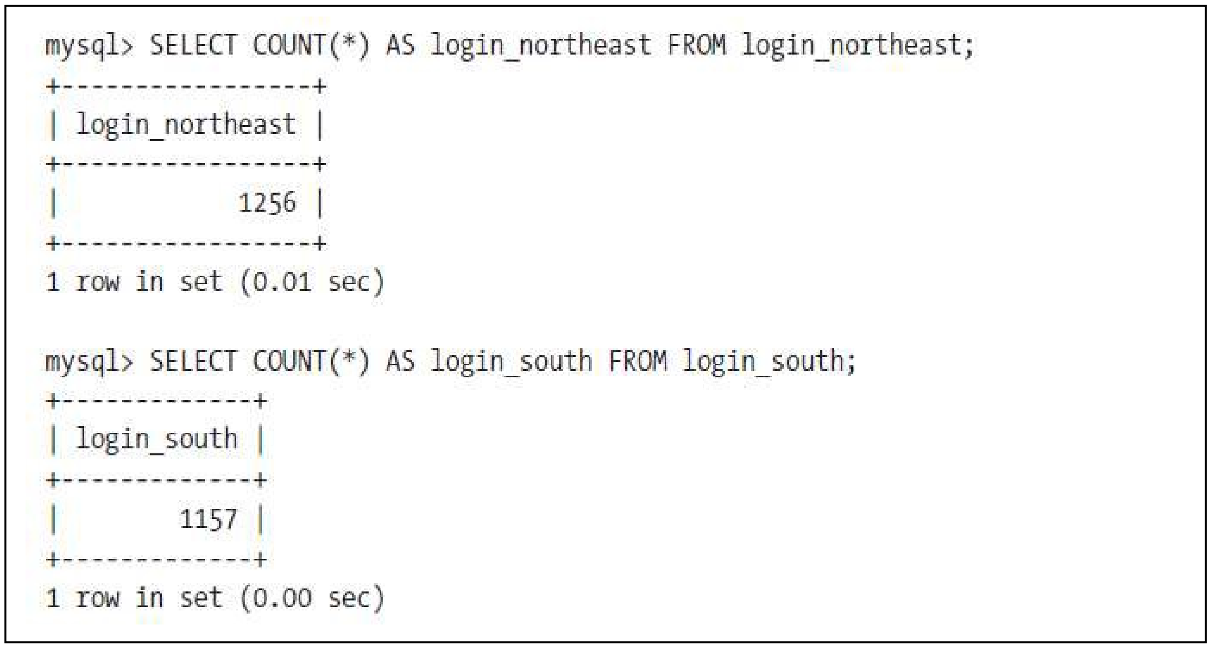
The return from login_archive() tells us that 3,757 of our 10,000 records were archived. To verify that records were moved into the appropriate place, we can run the queries to checkthe counts of each table, as shown in Listing 7.6.

**Listing 7-6. Record Counts After Calling archive_login()**



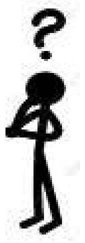
```
mysql> SELECT COUNT(*) AS login FROM login;
+-------+
| login |
+-------+
|  6243 |
+-------+
1 row in set (0.24 sec)

mysql> SELECT COUNT(*) AS login_northwest FROM login_northwest;
+-----------------+
| login_northwest |
+-----------------+
|            1344 |
+-----------------+
1 row in set (0.01 sec)
```

**Listing 7-6. Continuation**



```
mysql> SELECT COUNT(*) AS login_northeast FROM login_northeast;
+-----------------+
| login_northeast |
+-----------------+
|            1256 |
+-----------------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) AS login_south FROM login_south;
+-------------+
| login_south |
+-------------+
|        1157 |
+-------------+
1 row in set (0.00 sec)
```

The output from the counts on the tables indicates that some of the login table's records were moved to the three regional archive tables, and that there are still 6,243 records left. A look at the login table reveals that the remaining records are from logins from the past seven days, as we anticipated (Ramakrishhan, Raghu and Gehrke,Johannes,2018).



## Assessment Task

Based from your understanding, write a brief discussion and applications of cursors in MySQL. (2-3 paragraph)

**Note: Your answer will be tested with Plagiarism checker.**

## Summary

With the ability to add cursors into stored procedures and functions, you gain a powerful set of commands to process data, specifically in sets of results in your tables. Cursors are used in stored procedures, functions, and triggers to iterate through sets of data, assigning local variables to the field values in the data and using those values to perform logic or additional SQL statements. As with all database tools, you must carefully consider how cursors meet the needs of your users and application and determine whether using them within a procedure or function will provide the best solution to the problem you are attempting to solve. The cursor syntax is simple and easy to add into the syntax for stored routines in thedatabase. With just four additional statements, MySQL keeps focused on doing the most withthe least (Ramakrishhan, Raghu and Gehrke,Johannes,2018).

## References

Ramakrishhan, Raghu and Gehrke,Johannes (2018) ® Database Management Systems. McGraw Hill

Gupta, G. K., (2020) SQL for beginners: A guide to study SQL programming and database management systems Database management systems

- END OF THE MIDTERM MODULE -

CHECK YOUR EXAM SCHEDULE FOR THIS COURSE.

DO NOT FORGET TO TAKE THE EXAM AS SCHEDULED.

THANK YOU AND GOD BLESS