Merit-Quality-Excellence

IBA Institute of Emerging Technologies Khairpur

DATABASE MANAGEMENT SYSTEM

LAB No: 09

Instructor: Marina Gul

Objective of Lab No. 09:
After performing lab 09, students will be able to:

- To learn Stored Procedures
- To learn Stored Functions

Stored procedure

A stored routine is either a procedure or a function. Stored routines are created with CREATE PROCEDURE and CREATE FUNCTION statements. A procedure is invoked using a CALL statement, and can only pass back values using output variables. A function can be called from inside a statement just like any other function (that is, by invoking the function's name), and can return a scalar value. Stored routines may call other stored routines.

Compound Statement:

A compound statement is a block that can contain other blocks; declarations for variables, condition handlers, and cursors; and flow control constructs such as loops and conditional tests.

BEGIN ... END compound statement and other statements that can be used in the body of stored programs: Stored procedures and functions, triggers, and events. These objects are defined in terms of SQL code that is stored on the server for later invocation

Syntax:

```
BEGIN

[statement_list]

END
```

BEGIN ... END syntax is used for writing compound statements, which can appear within stored programs (stored procedures and functions, triggers, and events).

A compound statement can contain multiple statements, enclosed by the BEGIN and END keywords. statement_list represents a list of one or more statements, each terminated by a semicolon (;) statement delimiter. The statement_list itself is optional, so the empty compound statement (BEGIN END) is legal.

BEGIN ... END blocks can be nested.

Use of multiple statements requires that a client is able to send statement strings containing the **;** statement delimiter. In the mysql command-line client, this is handled with the delimiter command. Changing the **;** end-of-statement delimiter (for example, to //) permit ; to be used in a program body.

Stored Procedures:

The following SELECT statement returns all rows in the table customers from the database.

```
SELECT  customerName, city, state,  postalCode, country
FROM  customers
ORDER BY customerName;
```

If you want to save this query on the database server for execution later, one way to do it is to use a stored procedure.

The following CREATE PROCEDURE statement creates a new stored procedure that wraps the query above:

```
DELIMITER $$
CREATE PROCEDURE GetCustomers()
BEGIN
SELECT customerName, city, state, postalCode, country
FROM customers
ORDER BY customerName;
END$$
DELIMITER ;
```

By definition, a stored procedure is a segment of declarative SQL statements stored inside the MySQL Server. In this example, we have just created a stored procedure with the name GetCustomers().

Once you save the stored procedure, you can invoke it by using the CALL statement: And the statement returns the same result as the query.

```
CALL GetCustomers();
```

The first time you invoke a stored procedure, MySQL looks up for the name in the database catalog, compiles the stored procedure's code, place it in a memory area known as a cache, and execute the stored procedure.

If you invoke the same stored procedure in the same session again, MySQL just executes the stored procedure from the cache without having to recompile it.

A stored procedure can have parameters so you can pass values to it and get the result back. For example, you can have a stored procedure that returns customers by country and city. In this case, the country and city are parameters of the stored procedure.

A stored procedure may contain control flow statements such as IF, CASE, and LOOP that allow you to implement the code in the procedural way.

A stored procedure can call other stored procedures or stored functions, which allows you to modulize your code.

To create a new stored procedure, you use the CREATE PROCEDURE statement.
Here is the basic syntax of the CREATE PROCEDURE statement:

```
DELIMITER //
CREATE PROCEDURE procedure_name(parameter_list)
BEGIN
statements;
END //
DELIMITER;
```

The first and last DELIMITER commands are not a part of the stored procedure. The first DELIMITER command changes the default delimiter to // and the last DELIMITER command changes the delimiter back to the default one which is semicolon (;).
In given syntax:

- First, specify the name of the stored procedure that you want to create after the CREATE PROCEDURE keywords.
- Second, specify a list of comma-separated parameters for the stored procedure in parentheses after the procedure name.
- Third, write the code between the BEGIN END block. The above example just has a simple SELECT statement. After the END keyword, you place the delimiter character to end the procedure statement.

DELIMITER
When you write SQL statements, you use the semicolon (;) to separate two statements like the following example:

```
SELECT * FROM products;
SELECT * FROM customers;
```

A MySQL client program such as MySQL Workbench or mysql program uses the (;) delimiter to separate statements and executes each statement separately.
A stored procedure, however, consists of multiple statements separated by a semicolon (;).
If you use a MySQL client program to define a stored procedure that contains semicolon characters, the MySQL client program will not treat the whole stored procedure as a single statement, but many statements.

Therefore, you must redefine the delimiter temporarily so that you can pass the whole stored procedure to the server as a single statement.

To redefine the default delimiter, you use the DELIMITER command:

```
DELIMITER delimiter_character;
```

The delimiter_character may consist of a single character or multiple characters e.g., // or $$. However, you should avoid using the backslash (\) because this is the escape character in MySQL.

For example, this statement changes the delimiter to //:

```
DELIMITER //
```

Once change the delimiter, you can use the new delimiter to end a statement as follows:

```
DELIMITER //
SELECT * FROM customers //
SELECT * FROM products //
```
To change the delimiter back to semicolon, you use this statement:
```
DELIMITER ;
```

Using MySQL DELIMITER for stored procedures

A stored procedure typically contains multiple statements separated by semicolon (;). To use compile the whole stored procedure as a single compound statement, you need to temporarily change the delimiter from the semicolon (;) to anther delimiters such as $$ or //

```
DELIMITER $$
CREATE PROCEDURE sp_name()
BEGIN
-- statements
END $$
DELIMITER ;
```
In this code:

- First, change the default delimiter to $$
- Second, use (;) in the body of the stored procedure and $$ after the END keyword to end the stored procedure.
- Third, change the default delimiter back to a semicolon (;)

DROP Procedure

```
DROP PROCEDURE abc;
```

MySQL issue an error if procedure doesn't exist.
So, in order to avoid this error, you can use:

```
DROP PROCEDURE IF EXISTS abc;
```

Stored Procedure Parameters

Almost stored procedures that you develop require parameters. The parameters make the stored procedure more flexible and useful.
In MySQL, a parameter has one of three modes: `IN`, `OUT`, or `INOUT`.

`IN parameters`

IN is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure. In addition, the value of an IN parameter is protected. It means that even the value of the IN parameter is changed inside the stored procedure, its original value is retained after the stored procedure ends. In other words, the stored procedure only works on the copy of the IN parameter.

`OUT` parameters

The value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program. Notice that the stored procedure cannot access the initial value of the OUT parameter when it starts.

`INOUT` parameters

An INOUT parameter is a combination of IN and OUT parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter, and pass the new value back to the calling program.

Defining a parameter

Here is the basic syntax of defining a parameter in stored procedures:

```
[IN | OUT | INOUT] parameter_name datatype[(length)]
```

In this syntax,

- First, specify the parameter mode, which can be IN, OUT or INOUT, depending on the purpose of the parameter in the stored procedure.
- Second, specify the name of the parameter. The parameter name must follow the naming rules of the column name in MySQL.
- Third, specify the data type and maximum length of the parameter.

Stored Procedure Parameter Examples

The IN parameter example

The following example creates a stored procedure that finds all offices that locate in a country specified by the input parameter countryName:

```
DELIMITER //
CREATE PROCEDURE GetOfficeByCountry(
IN countryName VARCHAR(255)
)
BEGIN
SELECT *
FROM offices
WHERE country = countryName;
END //
DELIMITER ;
CALL GetOfficeByCountry('USA');
```

The OUT parameter example

The following stored procedure returns the number of orders by order status.

```
DELIMITER $$
CREATE PROCEDURE GetOrderCountByStatus (
IN orderStatus VARCHAR(25),
OUT total INT
)
BEGIN
SELECT COUNT(orderNumber) INTO total
FROM orders
WHERE status = orderStatus;
END$$
DELIMITER ;
```

The stored procedure GetOrderCountByStatus() has two parameters:

- orderStatus : is the IN parameter specifies the status of orders to return.
- total : is the OUT parameter that stores the number of orders in a specific status.

To find the number of orders that already shipped, you call GetOrderCountByStatus and pass the order status as of Shipped, and also pass a session variable (@total) to receive the return value.

```
CALL GetOrderCountByStatus('Shipped',@total);
SELECT @total;
```

The INOUT parameter example

The following example demonstrates how to use an INOUT parameter in the stored procedure.

```
DELIMITER $$
CREATE PROCEDURE SetCounter(
INOUT counter INT,
IN inc INT
)
BEGIN
SET counter = counter + inc;
END$$
DELIMITER ;
```

In this example, the stored procedure SetCounter() accepts one INOUT parameter (counter) and one IN parameter (inc). It increases the counter (counter) by the value of specified by the inc parameter.
These statements illustrate how to call the SetSounter stored procedure:

```
SET @counter = 1;
CALL SetCounter(@counter,1); -- 2
CALL SetCounter(@counter,1); -- 3
CALL SetCounter(@counter,5); -- 8
SELECT @counter; -- 8
```

Declare Statement
The DECLARE statement is used to define various items local to a program:

- `Local variables.`
- `Conditions and handlers.`

- Cursors.

DECLARE is permitted only inside a BEGIN ... END compound statement and must be at its start, before any other statements.

Declarations must follow a certain order. Cursor declarations must appear before handler declarations. Variable and condition declarations must appear before cursor or handler declarations.

Variables in Stored Programs
System variables and user-defined variables can be used in stored programs, just as they can be used outside stored-program context. In addition, stored programs can use DECLARE to define local variables, and stored routines (procedures and functions) can be declared to take parameters that communicate values between the routine and its caller.

- To declare local variables, use the DECLARE statement,
- Variables can be set directly with the SET statement.
- Results from queries can be retrieved into local variables

To declare a variable inside a stored procedure, you use the DECLARE statement as follows:

```
DECLARE variable_name datatype(size) [DEFAULT default_value];
```

In this syntax:

- First, specify the name of the variable after the DECLARE keyword. The variable name must follow the naming rules of MySQL table column names.
- Second, specify the data type and length of the variable. A variable can have any MySQL data types such as INT, VARCHAR , and DATETIME.
- Third, assign a variable a default value using the DEFAULT option. If you declare a variable without specifying a default value, its value is NULL.

The following example declares a variable named totalSale with the data type DEC(10,2) and default value 0.0 as follows:

```
DECLARE totalSale DEC(10,2) DEFAULT 0.0;
```

MySQL allows you to declare two or more variables that share the same data type using a single DECLARE statement. The following example declares two integer variables x and y, and set their default values to zero.

```
DECLARE x, y INT DEFAULT 0;
```

Assigning Variable
Once a variable is declared, it is ready to use. To assign a variable a value, you use the SET statement:

```
SET variable_name = value;
```

For example:

```
DECLARE total INT DEFAULT 0;
SET total = 10;
```

The value of the total variable is 10 after the assignment.
In addition to the SET statement, you can use the `SELECT INTO` statement to assign the result of a query to a variable as shown in the following example:

```
DECLARE productCount INT DEFAULT 0;
SELECT COUNT(*)
INTO productCount
FROM products;
```

In this example:

- First, declare a variable named productCount and initialize its value to 0.
- Then, use the `SELECT INTO` statement to assign the productCount variable the number of products selected from the products table.

Variable Scopes:

A variable has its own scope that defines its lifetime. If you declare a variable inside a stored procedure, it will be out of scope when the END statement of stored procedure reaches.
When you declare a variable inside the block BEGIN END, it will be out of scope if the END is reached.
MySQL allows you to declare two or more variables that share the same name in different scopes. Because a variable is only effective in its scope. However, declaring variables with the same name in different scopes is not good programming practice.
A variable whose name begins with the @ sign is a session variable. It is available and accessible until the session ends.

Putting it all together

```
DELIMITER $$
CREATE PROCEDURE GetTotalOrder()
BEGIN
DECLARE totalOrder INT DEFAULT 0;
SELECT COUNT(*)
INTO totalOrder

FROM orders;
SELECT totalOrder;
END$$
DELIMITER ;
CALL GetTotalOrder();
```

2. Stored Function

A stored function is a special kind stored program that returns a single value. Typically, you use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs.
Different from a stored procedure, you can use a stored function in SQL statements wherever an expression is used. This helps improve the readability and maintainability of the procedural code.
To create a stored function, you use the CREATE FUNCTION statement.

Syntax:

```
DELIMITER $$
CREATE FUNCTION function_name(
param1,
param2,…
)
RETURNS datatype
[NOT] DETERMINISTIC
BEGIN
-- statements
END $$
DELIMITER ;
```

In this syntax:

First, specify the name of the stored function that you want to create after CREATE FUNCTION keywords.

Second, list all parameters of the stored function inside the parentheses followed by the function name. By default, all parameters are the IN parameters. You cannot specify IN , OUT or INOUT modifiers to parameters

Third, specify the data type of the return value in the RETURNS statement, which can be any valid MySQL data types.

Fourth, specify if a function is deterministic or not using the DETERMINISTIC keyword.
A deterministic function always returns the same result for the same input parameters whereas a non-deterministic function returns different results for the same input parameters.

If you don't use DETERMINISTIC or NOT DETERMINISTIC, MySQL uses the NOT DETERMINISTIC option by default.

Fifth, write the code in the body of the stored function in the BEGIN END block. Inside the body section, you need to specify at least one RETURN statement. The RETURN statement returns a value to the calling programs. Whenever the RETURN statement is reached, the execution of the stored function is terminated immediately.

Example:

```
DELIMITER $$
CREATE FUNCTION CustomerLevel(
credit DECIMAL(10,2)
)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
DECLARE customerLevel VARCHAR(20);
IF credit > 50000 THEN
SET customerLevel = 'PLATINUM';
ELSEIF (credit >= 50000 AND
credit <= 10000) THEN
SET customerLevel = 'GOLD';
ELSEIF credit < 10000 THEN
SET customerLevel = 'SILVER';
END IF;
-- return the customer level
RETURN (customerLevel);
END$$
DELIMITER ;
```

Calling a stored function in an SQL statement

The following statement uses the CustomerLevel stored function:

```
SELECT
customerName,
CustomerLevel(creditLimit)
FROM
customers
ORDER BY
customerName;
```

Calling a stored function in a stored procedure

The following statement creates a new stored procedure that calls the CustomerLevel() stored function:

```
DELIMITER $$
CREATE PROCEDURE GetCustomerLevel(
IN customerNo INT,
OUT customerLevel VARCHAR(20)
)
BEGIN
DECLARE credit DEC(10,2) DEFAULT 0;
-- get credit limit of a customer
SELECT
creditLimit
INTO credit
FROM customers
WHERE
customerNumber = customerNo;
```

```
-- call the function
SET customerLevel = CustomerLevel(credit);
END$$
DELIMITER ;
```

The following illustrates how to call the GetCustomerLevel() stored procedure:

```
CALL GetCustomerLevel(-131,@customerLevel);
SELECT @customerLevel;
```

It’s important to notice that if a stored function contains SQL statements that query data from tables, then you should not use it in other SQL statements; otherwise, the stored function will slow down the speed of the query.

In this tutorial, you have learned how to create a stored function to encapsulate the common formula or business rules.

Drop Function

```
DROP FUNCTION IF EXISTS functionName;
```

Lab Task(s):

Exercise

1. Create a stored procedure **DISPLAY** without parameters. The procedure must display empno, ename and salary of all the employees of DEPTNO = 10.

2. Create a stored procedure **DISPLAY2** with parameters. It must take DEPTNO as an input and must return the DNAME and TOTAL SALARY of the input department number.

3. Create a stored procedure **DISPLAY3** with parameters. It must take DEPTNO as an input and must return the DNAME, SMALLEST and HIGHEST SALARIES of the input department number. DISPLAY3 must also display empno,ename,total salary (sal+comm) of all the employees of the input department number.

4. Create a stored function **MANAGER** without input parameters. It must return the total salary of all the managers in the EMP.

5. Create a stored function **MANAGER2** with parameters. It must take empno as an input and must return its manager name.

Write a SELECT statement to display all employees’ names and their manager names. Manager names must be displayed using MANAGER2 stored function.

6. Create a stored function **MANAGER3** with parameters. It must take MANAGER NAME as an input and must return average salary of its employees.

END