Lab 09
Instructor - Masood Iqbal @ ned/hpcc

Objective(s)

- Learn what is store procedure
- How to create and use store procedure
- SQL IF statement

Store Procedure

A stored procedure is a group of SQL statements that form a logical unit and perform a specific task. They are used to encapsulate a set of operations or queries to execute on a database server. Stored procedures can be compiled and executed with different parameters and results, and they can have any combination of input, output, and input/output parameters. A stored procedure can be invoked by triggers, other stored procedures, and applications.

Advantages of Store procedure

Reduce network traffic: An operation requiring hundreds of lines of SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.
Caching query plan: The first time the store procedure is executed, Database Server creates an execution plan, which is cached for reuse.
Faster Execution: If the operation requires a large amount of SQL code that is performed repetitively, stored procedures can be faster. They are parsed and optimized when they are first executed, and a compiled version of the stored procedure remains in a memory cache for later use. This means the stored procedure does not need to be reparsed and re-optimized with each use, resulting in much faster execution times.
Security: Users can be granted permission to execute a stored procedure even if they do not have permission to execute the procedure's statements directly.
SQL injection attacks: properly written inline SQL can defend against attacks, but store procedure is better for this protection.

Create Store Procedure

Each stored program contains a body that consists of an SQL statement. This statement may be a compound statement made up of several statements separated by semicolon (;) characters.

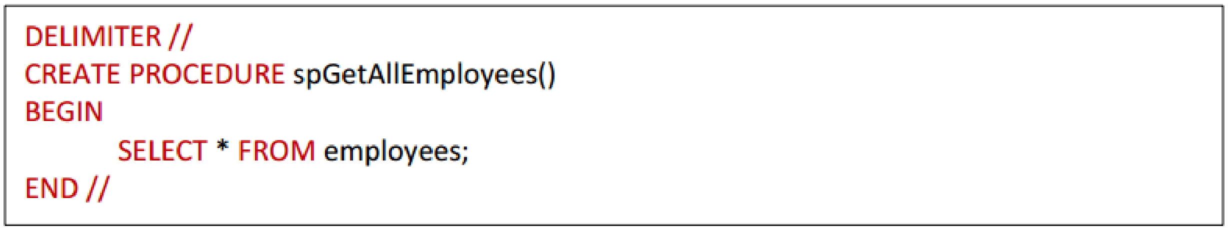
Syntax

```
CREATE PROCEDURE spName()
BEGIN
        [SQL statements]
END
```

If you use the MySQL client program to define a stored program containing semicolon characters, a problem arises. By default, MySQL itself recognizes the semicolon as a statement delimiter, so you must redefine the delimiter temporarily to cause MySQL to pass the entire stored program definition to the server.

```
DELIMITER //
CREATE PROCEDURE spName()
BEGIN
        [SQL statements]
END //
```

Example



```
DELIMITER //
CREATE PROCEDURE spGetAllEmployees()
BEGIN
        SELECT * FROM employees;
END //
```

- We use the **CREATE PROCEDURE** statement to create a new stored procedure. We Specify the name of stored procedure after the **CREATE PROCEDURE** statement. In this Case, the name of the stored procedure is spGetAllEmployees. We put the parentheses after the name of the stored procedure.
- The section between **BEGIN** and **END** is called the body of the stored procedure. You put the declarative SQL statements in the body to handle business logic. In this stored procedure, we use a simple **SELECT** statement to query data from the employees table.

Execute Store Procedure

To run a store procedure in MySQL use command **CALL** and then store procedure name.

Syntax

```
CALL StoreProcedureName();
CALL spGetAllEmployees();           -- Select All Employees
```

Stored Procedure Variables

A variable is a named data object whose value can change during the stored procedure execution. We typically use the variables in stored procedures to hold the immediate results. These variables are local to the stored procedure. You must declare a variable before you can use it.

Declaring variables

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

Syntax

```
DECLARE variable_name datatype(size) DEFAULT default_value;
DECLARE X INT DEFAULT 0;     -- Example
```

- First, you specify the variable name after the DECLARE keyword. The variable name must follow the naming rules of MySQL table column names.
- Second, you specify the data type of the variable and its size. A variable can have any MySQL data types such as INT, VARCHAR, DATETIME, etc.
- Third, when you declare a variable, its initial value is NULL. You can assign the variable a default value using the DEFAULT keyword.

MySQL allows you to declare two or more variables that share the same data type using a single Declare statement as following.
Example: **DECLARE** X, Y **INT DEFAULT** 0;

Assigning variables

Once you declared a variable, you can start using it. To assign a variable another value, you use can SET statement.

Syntax

```
SET variable_name = value;
SET X = 5;        -- Example: Assign a value to x variable
```

Besides the **SET** statement, you can use the **SELECT INTO** statement to assign the result of a query, which returns a scalar value, to a variable. See the following example:

Example

```
DECLARE max_salary INT DEFAULT 0;
SELECT MAX(salary) INTO max_salary FROM employees;
```

Variables scope

- 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 reached.
- If you declare a variable inside **BEGIN END** block, it will be out of scope if the **END** is reached. You can declare two or more variables with the same name in different scopes because a variable is only effective in its own scope. However, declaring variables with the same name in different scopes is not good programming practice.
- A variable that begins with the @ sign is session variable. It is available and accessible until the session ends.

Stored procedure 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** – 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** – an **INOUT** parameter is the 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.

Syntax

```
CREATE PROCEDURE spNoParam():        -- Parameter list is empty
CREATE PROCEDURE spInParam(IN varname DataType):         -- One input parameter, IN is optional
CREATE PROCEDURE spOutParam(OUT varname DataType):      -- One output parameter
CREATE PROCEDURE spInOutParam(INOUT varname DataType):  -- One parameter, which is both
                                                           Input and Output
```

No parameter example

```
DELIMITER //
CREATE PROCEDURE spGetAllEmployees()
BEGIN
        SELECT * FROM employees;
END //
```

```
CALL spGetAllEmployees();
```

IN Example

```
DELIMITER //
CREATE PROCEDURE spGetEmployeeById(IN emp_id INT)
BEGIN
        SELECT  employee_id, first_name, salary FROM employees
        WHERE  employee_id  = emp_id;
END//
```

```
CALL spGetEmployeeById(100);
```

```
DELIMITER //
CREATE PROCEDURE spGetEmployeeByIdAndName(IN emp_id INT, IN emp_name VARCHAR(255))
BEGIN
        SELECT  employee_id, first_name, salary FROM employees
        WHERE  employee_id  = emp_id AND first_name = emp_name;
END //
```

```
CALL spGetEmployeeByIdAndName (100, ‘Steven’);
```

OUT Example

```
DELIMITER //
CREATE PROCEDURE spOutTotalEmp(OUT total_employees INT)
BEGIN
        SELECT COUNT(employee_id) INTO total_employees FROM employees;
END //
```

```
CALL spOutTotalEmp(@totalEmp);
SELECT @totalEmp;
```

INTOUT Example

```
DELIMITER //
CREATE PROCEDURE spUpdateSal(IN emp_id INT, OUT previous_salary DECIMAL(8,2),
                                    INOUT current_salary DECIMAL(8,2))
BEGIN
        SELECT salary INTO previous_salary FROM employees
        WHERE employee_id = emp_id;
-- Update Salary of employee to new salary
        UPDATE employees
        SET salary = current_salary
        WHERE employee_id = emp_id;
END //
```

```
SET @EmpId = 100;
SET @NewSal = 26000.00;
CALL spUpdateSal(@EmpId, @PreSal, @NewSal);
SELECT @PreSal AS 'Previous Salary', @NewSal AS 'NEW Salary';
```

IF statement

The **MySQL** **IF** statement allows you to execute a set of **SQL** statements based on a certain condition or value of an expression. An expression can return one of three values **TRUE FALSE**, or **NULL**.

Syntax

```
IF expression THEN
  Statements;
END IF;
```

If the expression evaluates to **TRUE**, then the statements will be executed, otherwise, the control is passed to the next statement following the **END IF**.

IF ELSE

In case you want to execute statements when the expression evaluates to **FALSE**, you use the **IF ELSE** statement as follows:

Syntax

```
IF expression THEN
        Statement(s);
ELSE
        Else statement(s);
END IF;
```

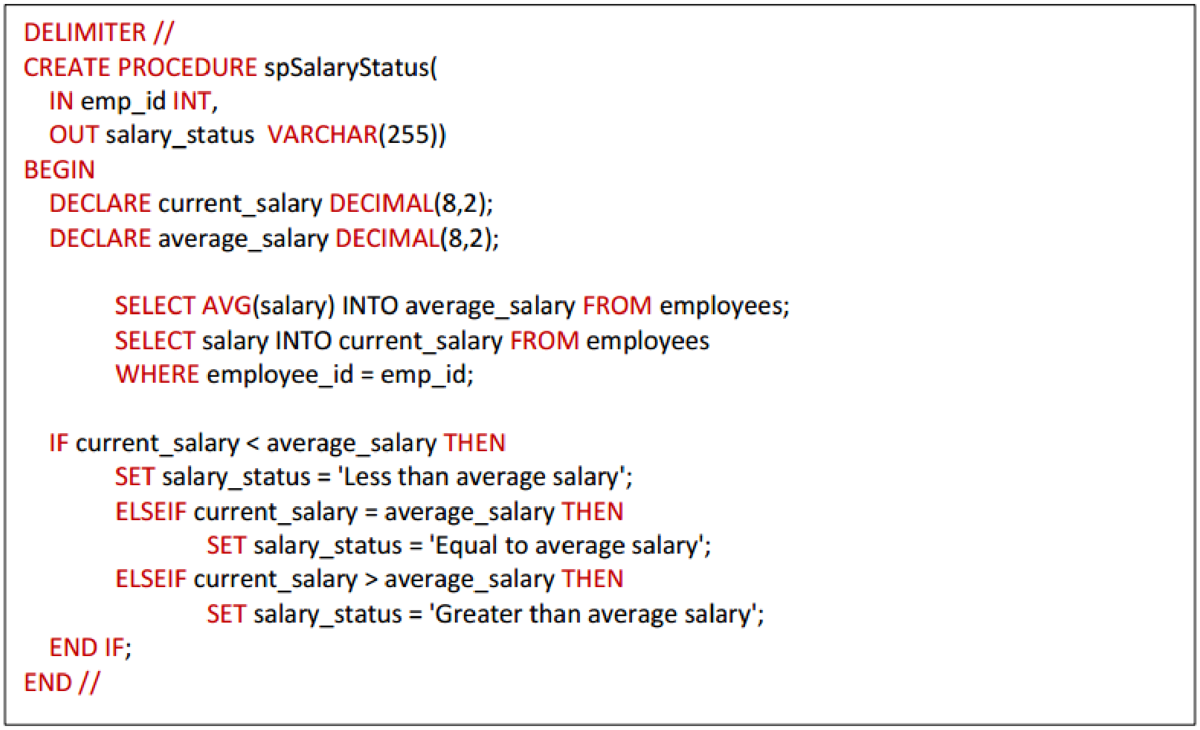
IF ELSEIF ELSE

If you want to execute statements conditionally based on multiple expressions, you use the **IF ELSEIF ELSE** statement as follows:

Syntax

```
IF expression THEN
        Statement(s);
ELSEIF expression THEN
        Elseif statement(s);
…
ELSE
        Else statement(s);
END IF;
```

Example



```
DELIMITER //
CREATE PROCEDURE spSalaryStatus(
   IN emp_id INT,
   OUT salary_status  VARCHAR(255))
BEGIN
   DECLARE current_salary DECIMAL(8,2);
   DECLARE average_salary DECIMAL(8,2);

           SELECT AVG(salary) INTO average_salary FROM employees;
           SELECT salary INTO current_salary FROM employees
           WHERE employee_id = emp_id;

   IF current_salary < average_salary THEN
           SET salary_status = 'Less than average salary';
           ELSEIF current_salary = average_salary THEN
                      SET salary_status = 'Equal to average salary';
           ELSEIF current_salary > average_salary THEN
                      SET salary_status = 'Greater than average salary';
   END IF;
END //
```

```
SET @EmpId = '100';
CALL spSalaryStatus(@EmpId, @SalaryStatus);
SELECT @SalaryStatus AS 'Salary Status of Employee';
```

Drop store procedure

You can a drop a store procedure by using the command below.

```
DROP PROCEDURE IF EXISTS sprocName;
```

LAB TASK

1. Create store procedure with no parameter
2. Create store procedure with IN parameter (optional two or more parameters)
3. Create store procedure with OUT parameter (optional two or more parameters)
4. Create store procedure with INOUT parameter ()
5. USE IF ELSEIF, ELSE
6. Drop any one store procedure

Note:

- Store Procedure perform complex tasks(not use simple queries)
- CALL store procedure e.g. CREATE then CALL for each task.