Stored Procedure

What is stored procedure?

- A stored procedure is a group of one or more database statements stored in the database’s data dictionary and called from either a remote program, another stored procedure, or the command line.

- A procedure or function is a schema object that logically groups a set of SQL and other PL/SQL programming language statements together to perform a specific task. Procedures and functions are created in a user's schema and stored in a database for continued use.

- A stored procedure or function is associated with a particular database. This has several implications:
- When the routine is invoked, an implicit USE *db_name* is performed (and undone when the routine terminates). USE statements within stored routines are not permitted.
- You can qualify routine names with the database name.
- This can be used to refer to a routine that is not in the current database.
- For example, to invoke a stored procedure p or function f that is associated with the test database, you can say CALL test.p() or test.f().
- When a database is dropped, all stored routines associated with it are dropped as well.
- Stored functions cannot be recursive.

- MySQL 5.6 supports "routines" and there are two kinds of routines :
- stored procedures which you call, or
- functions whose return values you use in other SQL statements the same way that you use pre-installed MySQL functions like pi().
- The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL statement.

Why Stored Procedures?

- Stored procedures are fast.
  - MySQL server takes some advantage of caching, just as prepared statements do.
  - The main speed gain comes from reduction of network traffic.
  - If you have a repetitive task that requires checking, looping, multiple statements, and no user interaction, do it with a single call to a procedure that's stored on the server.
- Stored procedures are portable.
  - When you write your stored procedure in SQL, you know that it will run on every platform that MySQL runs on, without obliging you to install an additional runtime-environment package, or set permissions for program execution in the operating system, or deploy different packages if you have different computer types. That's the advantage of writing in SQL rather than in an external language like Java or C or PHP.

Create Procedure

- By default, a procedure is associated with the default database (currently used database).
- To associate the procedure with a given database, specify the name as **database_name.stored_procedure_name** when you create it.

- **Syntax:**

```
CREATE [DEFINER = { user | CURRENT_USER }]
PROCEDURE sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
proc_parameter: [ IN | OUT | INOUT ] param_name type
```

- type: Any valid MySQL data type
- characteristic: COMMENT 'string' | LANGUAGE SQL | [NOT] DETERMINISTIC | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } | SQL SECURITY { DEFINER | INVOKER }
- routine_body: Valid SQL routine statement

- **Check the MySQL version:**
- SELECT VERSION();

- **Check the privileges of the current user:**
- SHOW PRIVILEGES;

- **Select a database:**
- USE TEST;

- **Pick a Delimiter**

Pick a Delimiter

- The delimiter is the character or string of characters which is used to complete an SQL statement.
- By default we use semicolon (;) as a delimiter.
- But this causes problem in stored procedure because a procedure can have many statements, and everyone must end with a semicolon.
- So for your delimiter, pick a string which is rarely occur within statement or within procedure.
- Here we have used double dollar sign i.e. $$.
- To resume using ";" as a delimiter later, say "DELIMITER ; $$".

- **DELIMITER $$**

- **SELECT * FROM user $$**

- execute the following command to resume "**;**" as a delimiter :
- **DELIMITER ;**

CREATE PROCEDURE

- Here we have created a simple procedure called job_data, when we will execute the procedure it will display all the data from "jobs" tables.

- **DELIMITER $$**
- **CREATE PROCEDURE job_data()**
  SELECT * FROM JOBS; $$

Call a procedure

- The CALL statement is used to invoke a procedure that is stored in a DATABASE.

- Syntax :

CALL sp_name([parameter[,...]])

CALL sp_name[()]

- Stored procedures which do not accept arguments can be invoked without parentheses.
- Therefore CALL job_data() and CALL job_data are equivalent.
- To execute the procedure.

```
CALL job_data$$
```

SHOW CREATE PROCEDURE

- This statement is a MySQL extension.
- It returns the exact string that can be used to re-create the named stored procedure.
- Both statement require that you be the owner of the routine.

- Syntax :
- **SHOW CREATE PROCEDURE proc_name**

- Example
- **SHOW CREATE PROCEDURE job_data$$**

- There are some clauses in CREATE PROCEDURE syntax which describe the characteristics of the procedure.
- The clauses come after the parentheses, but before the body.
- These clauses are all optional.
- Here are the clauses :

```
characteristic:
COMMENT 'string'
| LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL
| NO SQL
| READS SQL DATA
| MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
```

- **COMMENT** :
- The COMMENT characteristic is a MySQL extension. It is used to describe the stored routine and the information is displayed by the SHOW CREATE PROCEDURE statements.
- **LANGUAGE :**
- The LANGUAGE characteristic indicates that the body of the procedure is written in SQL.
- **NOT DETERMINISTIC :**
- NOT DETERMINISTIC, is informational, a routine is considered "deterministic" if it always produces the same result for the same input parameters, and "not deterministic" otherwise.

- **CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA**
- **CONTAINS SQL :**
- CONTAINS SQL means there are no statements that read or write data, in the routine. For example statements SET @x = 1 or DO RELEASE_LOCK('abc'), which execute but neither read nor write data. This is the default if none of these characteristics is given explicitly.
- **NO SQL:**
- NO SQL means routine contains no SQL statements.
- **READS SQL DATA :**
- READS SQL DATA means the routine contains statements that read data (for example, SELECT), but not statements that write data.
- **MODIFIES SQL DATA :**
- MODIFIES SQL DATA means routine contains statements that may write data (for example, INSERT or DELETE).

- **SQL SECURITY { DEFINER | INVOKER }**
- SQL SECURITY, can be defined as either SQL SECURITY DEFINER or SQL SECURITY INVOKER to specify the security context; that is, whether the routine executes using the privileges of the account named in the routine DEFINER clause or the user who invokes it. This account must have permission to access the database with which the routine is associated. The default value is DEFINER. The user who invokes the routine must have the EXECUTE privilege for it, as must the DEFINER account if the routine executes in definer security context.

- All the above characteristics clauses have defaults.
- Following two statements produce same result :
- **CREATE PROCEDURE job_data()**

```
        SELECT * FROM JOBS; $$
```

- **CREATE PROCEDURE new_job_data()**

```
     COMMENT ''
    LANGUAGE SQL
    NOT DETERMINISTIC
    CONTAINS SQL
     SQL SECURITY DEFINER
     SELECT * FROM JOBS;
     $$
```

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
- Statement Label
- DECLAREVariables in Stored Programs
- Flow Control Statements
- Cursors
- Condition Handling

BEGIN ... END Compound-Statement Syntax

- BEGIN ... END block is used to write compound statements, i.e. when you need more than one statement within stored programs (e.g. stored procedures, functions, triggers, and events).
- Syntax :

```
[begin_label:]
BEGIN
        [statement_list]
END
[end_label])
```

- statement_list : It represents one or more statements terminated by a semicolon(;). The statement_list itself is optional, so the empty compound statement BEGIN END is valid.

Label Statement

- Labels are permitted for BEGIN ... END blocks and for the LOOP, REPEAT, and WHILE statements.
- Syntax :

[*begin_label*:]

BEGIN

[*statement_list*]

END

[*end_label*]

- Label use for those statements which follows following rules:
    - begin_label must be followed by a colon
    - begin_label can be given without end_label. If end_label is present, it must be the same as begin_label
    - end_label cannot be given without begin_label.
    - Labels at the same nesting level must be distinct
    - Labels can be up to 16 characters long.

Declare Statement

- The DECLARE statement is used to define various items local to a program, for example local variables, conditions and handlers, cursors.
- DECLARE is used only inside a BEGIN ... END compound statement and must be at its start, before any other statements.
- Declarations follow the following 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.
- Stored programs 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.

- **Declare a Variable:**
- DECLARE var_name [, var_name] ... type [DEFAULT value]
- To provide a default value for a variable, include a DEFAULT clause.
- The value can be specified as an expression; it need not be constant.
- If the DEFAULT clause is missing, the initial value is NULL.

Local variables

- Local variables are declared within stored procedures and are only valid within the BEGIN…END block where they are declared.

- Local variables can have any SQL data type.

- The following example shows the use of local variables in a stored procedure.

Example

◦
```
CREATE PROCEDURE proc_Variables()
BEGIN
        DECLARE a INT DEFAULT 10;
        DECLARE b, c INT;
        SET a = a + 100;
        SET b = 2;
        SET c = a + b;
        BEGIN
                DECLARE c INT;
                SET c = 5;
                SELECT a, b, c;
        END;
        SELECT a, b, c;
END$$
```
◦
```
CALL proc_Variables()$$
```

User variables

- In MySQL stored procedures, user variables are referenced with an ampersand (@) prefixed to the user variable name (for example, @x and @y).

- The following example shows the use of user variables within the stored procedure :

Example

- CREATE PROCEDURE proc_User_Variables()

```
    BEGIN
        SET @x = 15;
        SET @y = 10;
        SELECT @x, @y, @x-@y;
    END$$
```

- CALL proc_User_Variables()$$

Procedure Parameters

- Syntax :
- CREATE

[DEFINER = { user | CURRENT_USER }]

PROCEDURE sp_name ([proc_parameter[,...]])

[characteristic ...] routine_body

proc_parameter: [IN | OUT | INOUT] param_name type

- We can divide the above CREATE PROCEDURE statement in the following ways :

1. CREATE PROCEDURE sp_name () ...
   - The parameter list is empty.

2. CREATE PROCEDURE sp_name ([IN] param_name type)...
   - IN parameter passes a value into a procedure.
   - The procedure might modify the value, but the modification is not visible to the caller when the procedure returns.

3. CREATE PROCEDURE sp_name ([OUT] param_name type)...
   - an OUT parameter passes a value from the procedure back to the caller.
   - Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.

4. CREATE PROCEDURE sp_name ([INOUT] param_name type)...
   - an INOUT parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.

- In a procedure, each parameter is an IN parameter by default. To specify otherwise for a parameter, use the keyword OUT or INOUT before the parameter name.

Parameter IN example

- In the following procedure, we have used a IN parameter 'var1' (type integer) which accept a number from the user.
- Within the body of the procedure, there is a SELECT statement which fetches rows from 'jobs' table and the number of rows will be supplied by the user.

```
CREATE PROCEDURE proc_IN(IN var1 INT)
        BEGIN
                SELECT * FROM jobs LIMIT var1;
        END$$
```

- **CALL proc_in(2)$$**
- **CALL proc_in(7)$$**

Parameter OUT example

- The following example shows a simple stored procedure that uses an OUT parameter.
- Within the procedure MySQL MAX() function retrieves maximum salary from MAX_SALARY of jobs table.

```
CREATE PROCEDURE my_proc_OUT (OUT highest_salary INT)
BEGIN
        SELECT MAX(MAX_SALARY) INTO highest_salary FROM JOBS;
END$$
```

```
CALL my_proc_OUT(@M)$$
SELECT @M$$
```

Parameter INOUT example

- The user will supply 'M' or 'F' through IN parameter (emp_gender) to count a number of male or female from user_details table.
- The INOUT parameter (mfgender) will return the result to a user.

- **CREATE PROCEDURE proc_INOUT (INOUT maxsalary INT, IN desig CHAR(30))**

BEGIN

    SELECT MAX_SALARY INTO maxsalary FROM jobs WHERE JOB_TITLE = desig;

END$$

- **CALL proc_INOUT(@C,'Stock Clerk')$$**
- **SELECT @C$$**

Flow Control Statements

- MySQL supports IF, CASE, ITERATE, LEAVE, LOOP, WHILE, and REPEAT constructs for flow control within stored programs.
- It also supports RETURN within stored functions.

If Statement

- The IF statement implements a basic conditional construct within a stored programs and must be terminated with a semicolon.
- There is also an IF() function, which is different from the IF statement.
- Here is the syntax of if statement :

IF *condition* THEN *statement(s)*

[ELSEIF *condition* THEN *statement(s)*] ...

[ELSE *statement(s)*]

END IF

```
◦ CREATE PROCEDURE proc_Compare_Salary(IN desig CHAR(30))
  BEGIN
      DECLARE progSal INT;
     DECLARE presSal INT;
     SELECT MIN_SALARY INTO progSal FROM jobs WHERE JOB_TITLE = desig;
     SELECT MIN_SALARY INTO presSal FROM jobs WHERE JOB_TITLE = 'president';
     SET presSal = presSal /2;
     SELECT presSal;
     SELECT progSal;
     IF (progSal < presSal)
     THEN
             SELECT 'You need an increment.';
     ELSEIF (progSal = presSal)
     THEN
             SELECT 'Salary is moderate';
     ELSE
             SELECT 'You are earning good';
     END IF;
  END$$
```

```
◦   CALL proc_Compare_Salary('Programmer')$$
```

Case Statement

- The CASE statement is used to create complex conditional construct within stored programs.
- The CASE statement cannot have an ELSE NULL clause, and it is terminated with END CASE instead of END.
- Here is the syntax :

```
CASE case_value
WHEN when_value THEN statement_list
[WHEN when_value THEN statement_list]
... [ELSE statement_list]
END CASE
```

or

```
CASE
WHEN search_condition THEN statement_list
[WHEN search_condition THEN statement_list]
 ... [ELSE statement_list]
 END CASE
```

- count the number of employees with following conditions from jobs table :
- MIN_SALARY > 10000
  MIN_SALARY < 10000
  MIN_SALARY = 10000

```
◦ CREATE PROCEDURE proc_Case (INOUT no_employees INT, IN
  salary INT)
BEGIN
CASE
        WHEN (salary>10000)
        THEN (SELECT COUNT(job_id) INTO no_employees FROM
jobs                    WHERE min_salary>10000);
        WHEN (salary<10000)
        THEN (SELECT COUNT(job_id) INTO no_employees FROM
jobs                    WHERE min_salary<10000);
        ELSE (SELECT COUNT(job_id) INTO no_employees FROM jobs
                        WHERE min_salary=10000);
        END CASE;
END$$
◦ CALL proc_Case(@C,10001)$$
◦ SELECT @C$$
```

- **Number of employees whose salary greater than 10000 :**
- **CALL proc_Case(@C,10001);**

- **Number of employees whose salary less than 10000 :**
- **CALL proc_Case(@C,9999);**

- **Number of employees whose salary equal to 10000 :**
- **CALL proc_Case(@C,10000);**

ITERATE Statement

- ITERATE means "start the loop again".
- ITERATE can appear only within LOOP, REPEAT, and WHILE statements.
- Syntax :

ITERATE label

LEAVE Statement

- LEAVE statement is used to exit the flow control construct that has the given label.
- If the label is for the outermost stored program block, LEAVE exits the program.
- LEAVE can be used within BEGIN ... END or loop constructs (LOOP, REPEAT, WHILE).
- Syntax :

LEAVE label

LOOP Statement

- LOOP is used to create repeated execution of the statement list.
- Syntax :

```
[begin_label:]
 LOOP
              statement_list
 END LOOP
[end_label]
```

- statement_list consists one or more statements, each statement terminated by a semicolon (;).
- The statements within the loop are repeated until the loop is terminated.
- Usually, LEAVE statement is used to exit the loop construct.
- Within a stored function, RETURN can also be used, which exits the function entirely.
- A LOOP statement can be labeled.

Example

- In the following procedure rows will be inserted in 'number' table until x is less than num (number supplied by the user through IN parameter). A random number will be stored every time.
- **CREATE PROCEDURE proc_LOOP (IN num INT)**

```
BEGIN
        DECLARE x INT;
        SET x = 0;
        loop_label: LOOP
                INSERT INTO number VALUES (rand());
                SET x = x + 1;
                IF x >= num
                THEN LEAVE loop_label;
                END IF;
        END LOOP;
END$$
```

- **CALL proc_LOOP(3)$$**
- **Select * from number$$**

CREATE TABLE numbers (n1 real(3,2));

REPEAT Statement

- The REPEAT statement executes the statement(s) repeatedly as long as the condition is true.
- The condition is checked every time at the end of the statements.

```
[begin_label:]
REPEAT
        statement_list
UNTIL search_condition
END REPEAT
[end_label]
```

- statement_list: List of one or more statements, each statement terminated by a semicolon(;).
- search_condition : An expression.
- A REPEAT statement can be labeled.
-

Example:

- In the following procedure user passes a number through IN parameter and make a sum of even numbers between 1 and that particular number.
- **CREATE PROCEDURE proc_REPEAT (IN n INT)**

```
BEGIN
        SET @sum = 0;
        SET @x = 1;
        REPEAT
                IF mod(@x, 2) = 0
                THEN SET @sum = @sum + @x;
                END IF;
                SET @x = @x + 1;
        UNTIL @x > n
        END REPEAT;
END $$
```

- **CALL proc_REPEAT(5)$$**
- **SELECT @sum$$**

RETURN Statement

- The RETURN statement terminates execution of a stored function and returns the value *expr* to the function caller.
- There must be at least one RETURN statement in a stored function.
- There may be more than one if the function has multiple exit points.
- Syntax :

RETURN expr

- This statement is not used in stored procedures or triggers.
- The LEAVE statement can be used to exit a stored program of those types.

WHILE Statement

- The WHILE statement executes the statement(s) as long as the condition is true.
- The condition is checked every time at the beginning of the loop. Each statement is terminated by a semicolon (;).
- Syntax:

```
[begin_label:]
WHILE search_condition
DO
        statement_list
END WHILE
[end_label]
```

Example:

- In the following procedure, a user passes a number through IN parameter and make a sum of odd numbers between 1 and that particular number.
- **CREATE PROCEDURE proc_WHILE(IN n INT)**

```
BEGIN
        SET @sum = 0;
        SET @x = 1;
        WHILE @x<n
        DO
                IF mod(@x, 2) <> 0
                THEN SET @sum = @sum + @x;
                END IF;
                SET @x = @x + 1;
        END WHILE;
END$$
```

- **CALL proc_WHILE(5)$$**
- **SELECT @sum$$**
- **CALL proc_WHILE(10)$$**
- **SELECT @sum$$**

ALTER PROCEDURE

- This statement can be used to change the characteristics of a stored procedure.
- More than one change may be specified in an ALTER PROCEDURE statement.
- However, you cannot change the parameters or body of a stored procedure using this statement; to make such changes, you must drop and re-create the procedure using DROP PROCEDURE and CREATE PROCEDURE.
- Syntax :

ALTER PROCEDURE proc_name [characteristic ...]characteristic: COMMENT 'string'

| LANGUAGE SQL

| { CONTAINS SQL

| NO SQL

| READS SQL DATA

| MODIFIES SQL DATA }

| SQL SECURITY { DEFINER | INVOKER }

- You must have the ALTER ROUTINE privilege for the procedure. By default, that privilege is granted automatically to the procedure creator.

Example

- **ALTER PROCEDURE proc_WHILE COMMENT 'Modify Comment';**
- **SHOW CREATE PROCEDURE proc_WHILE;**

DROP PROCEDURE

- This statement is used to drop a stored procedure or function.
- That is, the specified routine is removed from the server.
- You must have the ALTER ROUTINE privilege for the routine.

DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name

- The IF EXISTS clause is a MySQL extension.
- It prevents an error from occurring if the procedure or function does not exist.
- A warning is produced that can be viewed with SHOW WARNINGS.
- Example:
- **DROP PROCEDURE new_procedure;**
- **SHOW CREATE PROCEDURE new_procedure;**

Cursors

- A database cursor is a control structure that enables traversal over the records in a database.
- Cursors are used by database programmers to process individual rows returned by database system queries.
- Cursors enable manipulation of whole result sets at once.
- In this scenario, a cursor enables the rows in a result set to be processed sequentially.
- In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis.
- By using the same mechanics, an SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

- MySQL supports cursors inside stored programs.
- The syntax is as in embedded SQL.
- Cursors have these properties :
    - Asensitive:
        - The server may or may not make a copy of its result table
    - Read only:
        - Not updatable
    - Nonscrollable:
        - Can be traversed only in one direction and cannot skip rows
- Steps to use cursor:
  - Declare a cursor.
  - Open a cursor.
  - Fetch the data into variables.
  - Close the cursor when done.

- **Declare a cursor:**
- The following statement declares a cursor and associates it with a SELECT statement that retrieves the rows to be traversed by the cursor.
- DECLARE cursor_name CURSOR FOR select_statement

- **Open a cursor:**
- The following statement opens a previously declared cursor.
- OPEN cursor_name

- **Fetch the data into variables :**
- This statement fetches the next row for the SELECT statement associated with the specified cursor (which must be open) and advances the cursor pointer. If a row exists, the fetched columns are stored in the named variables. The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement.
- FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...

- **Close the cursor when done :**
- This statement closes a previously opened cursor. An error occurs if the cursor is not open.
- CLOSE cursor_name

Example:

- The procedure starts with three variable declarations. Incidentally, the order is important. First, declare variables. Then declare conditions. Then declare cursors. Then, declare handlers. If you put them in the wrong order, you will get an error message.
- **CREATE PROCEDURE proc_cursors(INOUT return_val INT)**

```
BEGIN
        DECLARE a,b INT;
        DECLARE cur_1 CURSOR FOR SELECT max_salary FROM jobs;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;
        OPEN cur_1;
                REPEAT
                        FETCH cur_1 INTO a;
                UNTIL b = 1
        END REPEAT;
        CLOSE cur_1;
        SET return_val = a;
END$$
```

- **CALL proc_cursors(@R)$$**
- **SELECT @R$$**

Show the list of procedures

- SHOW PROCEDURE STATUS;

- To list all stored procedures of the databases that you have the privilege to access

```
SHOW PROCEDURE STATUS;
```

- To list a stored procedure from a particular database

```
SHOW PROCEDURE STATUS WHERE db = ’databasename';
```

- To show stored procedures that have a particular pattern

```
SHOW PROCEDURE STATUS WHERE name LIKE '%product%’;
```

Error handling in Stored procedure

- When an error occurs inside a stored procedure, it is important to handle it appropriately, such as continuing or exiting the current code block's execution, and issuing a meaningful error message.

- **Declaring a handler**
- To declare a handler, you use the DECLARE HANDLER statement as follows:

```
DECLARE action HANDLER FOR condition_value statement;
```

-

- **Three type of Handler_Action:**
- CONTINUE
- EXIT
- UNDO

- **Type of Condition Value:**
- mysql_error_code
- sqlstate_value
- SQLWarning
- SQLException
- NotFound

Example

- CREATE OR REPLACE PROCEDURE testErrorCONT()

  BEGIN

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SELECT 'Error occured';

  INSERT INTO csd2204w18.customers(customer_id,cust_name) VALUES(3001,"test");

  SELECT *FROM csd2204w18.customers;

  END$$
- CALL testErrorCONT()$$

Example

- CREATE OR REPLACE PROCEDURE testErrorEXIT()

  BEGIN

  DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'Error occured';

  INSERT INTO csd2204w18.customers(customer_id,cust_name) VALUES(3001,"test");

  SELECT *FROM csd2204w18.customers;

  END$$
- CALL testErrorEXIT()$$

Class Activity

1. Create a stored procedure that accepts a string corresponding to four arithmetic operations and perform corresponding operation.

- CALL SP_Airthematic_Operation("Div",5,6);
- CALL SP_Airthematic_Operation("Mul",5,6);
- CALL SP_Airthematic_Operation("Sub",5,6);
- CALL SP_Airthematic_Operation("Add",5,6);

Example

```
CREATE OR REPLACE PROCEDURE SP_Airthematic_Operation
(caseValue Varchar(10),valueA Integer(10),valueB Integer(10))
   BEGIN
   CASE caseValue
          WHEN "Add" THEN SELECT (valueA+valueB) as Addition;
           WHEN "Sub" THEN SELECT (valueA-valueB) as Subtraction;
          WHEN "Mul" THEN SELECT (valueA*valueB) as Multiple;
          WHEN "Div" THEN SELECT (valueA/valueB) as Divide;
          ELSE
                    BEGIN
                    END;
   END CASE;
   END$$
```

References

- https://www.w3resource.com/mysql/mysql-procedure.php
- https://www.dbrnd.com/2015/05/mysql-error-handling/