A procedure (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database. A procedure has a name, a parameter list, and SQL statement(s). All most all relational database system supports stored procedure, MySQL 5 introduce stored procedure.

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.
- Stored procedures are always available as 'source code' in the database itself. And it makes sense to link the data with the processes that operate on the data.

Create Procedure

Following statements create a stored 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. Here is the complete syntax :

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
```

Before create a procedure we need some information which are described in this section :

Check the MySQL version:

Following command displays the version of MySQL :

```
mysql>SELECT VERSION();
+-----------+
| VERSION() |
+-----------+
| 5.6.12    |
+-----------+
1 row in set (0.00 sec)
```

Check the privileges of the current user:

CREATE PROCEDURE and CREATE FUNCTION require the CREATE ROUTINE privilege. They might also require the SUPER privilege, depending on the DEFINER value, as described later in this section. If binary logging is enabled, CREATE FUNCTION might require the SUPER privilege. By default, MySQL automatically grants the ALTER ROUTINE and EXECUTE privileges to the routine creator. This behavior can be changed by disabling the automatic_sp_privileges system variable.

```
mysql> SHOW PRIVILEGES;
+-----------------+---------------------------+-----------------------------------------------------------+
| Privilege       | Context                   | Comment                                                   |
+-----------------+---------------------------+-----------------------------------------------------------+
| Alter           | Tables                    | To alter the table                                        |
| Alter routine   | Functions,Procedures      | To alter or drop stored functions/procedures              |
| Create          | Databases,Tables,Indexes  | To create new databases and tables                        |
| Create routine  | Databases                 | To use CREATE FUNCTION/PROCEDURE                          |
| Create temporary| Databases                 | To use CREATE TEMPORARY TABLE                             |
| tables          |                           |                                                           |
| Create view     | Tables                    | To create new views                                       |
| Create user     | Server Admin              | To create new users                                       |
| Delete          | Tables                    | To delete existing rows                                   |
| Drop            | Databases,Tables          | To drop databases, tables, and views                      |
| Event           | Server Admin              | To create, alter, drop and execute events                 |
| Execute         | Functions,Procedures      | To execute stored routines                                |
| File            | File access on server     | To read and write files on the server                     |
| Grant option    | Databases,Tables,         | To give to other users those privileges you possess       |
|                 | Functions,Procedures      |                                                           |
| Index           | Tables                    | To create or drop indexes                                 |
| Insert          | Tables                    | To insert data into tables                                |
| Lock tables     | Databases                 | To use LOCK TABLES (together with SELECT privilege)       |
| Process         | Server Admin              | To view the plain text of currently executing queries     |
| Proxy           | Server Admin              | To make proxy user possible                               |
| References      | Databases,Tables          | To have references on tables                              |
| Reload          | Server Admin              | To reload or refresh tables, logs and privileges          |
| Replication     | Server Admin              | To ask where the slave or master servers are              |
| client          |                           |                                                           |
| Replication     | Server Admin              | To read binary log events from the master                 |
```

```
| slave          |                    |                                                |
| Select         | Tables             | To retrieve rows from table                    |
| Show databases | Server Admin       | To see all databases with SHOW DATABASES       |
| Show view      | Tables             | To see views with SHOW CREATE VIEW             |
| Shutdown       | Server Admin       | To shut down the server                        |
| Super          | Server Admin       | To use KILL thread, SET GLOBAL, CHANGE MASTER, etc. |
| Trigger        | Tables             | To use triggers                                |
| Create         | Server Admin       | To create/alter/drop tablespaces               |
| tablespace     |                    |                                                |
| Update         | Tables             | To update existing rows                        |
| Usage          | Server Admin       | No privileges - allow connect only             |
+----------------------+--------------------+------------------------------------------------------+
31 rows in set (0.00 sec)
```

Select a database:

Before creates a procedure we must select a database. Let see the list of databases and choose one of them.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| hr                 |
| mysql              |
| performance_schema |
| sakila             |
| test               |
| world              |
+--------------------+
7 rows in set (0.06 sec))
```

Now select the database 'hr' and list the tables :

```
mysql> USE hr;
Database changed
mysql> SHOW TABLES;
+--------------+
| Tables_in_hr |
+--------------+
| alluser      |
| departments  |
| emp_details  |
| job_history  |
| jobs         |
| locations    |
| regions      |
| user         |
| user_details |
+--------------+
```

9 rows in set (0.00 sec)

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. $$.You can use whatever you want. To resume using ";" as a delimiter later, say "DELIMITER ; $$". See here how to change the delimiter :

```
mysql> DELIMITER $$ ;
```

Now the default DELIMITER is "$$". Let execute a simple SQL command :

```
mysql> SELECT * FROM user $$
+----------+-----------+--------+
| userid   | password  | name   |
+----------+-----------+--------+
| scott123 | 123@sco   | Scott  |
| ferp6734 | dloeiu@&3 | Palash |
| diana094 | ku$j@23   | Diana  |
+----------+-----------+--------+
3 rows in set (0.00 sec)
```

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

```
mysql> DELIMITER ; $$
```

Example : MySQL 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.

```
mysql> DELIMITER $$ ;
mysql> CREATE PROCEDURE job_data()
            > SELECT * FROM JOBS;
          $$
Query OK, 0 rows affected (0.00 sec)
```

Explanation:

- CREATE PROCEDURE command creates the stored procedure.
- Next part is the procedure name. Here the procedure name is " job_data".

-- Procedure names are not case sensitive, so job_data and JOB_DATA are same.

-- You cannot use two procedures with the same name in the same database.

-- You can use qualified names of the form "database-name**.**procedure-name", for example "hr.job_data".

-- Procedure names can be delimited. If the name is delimited, it can contain spaces.

-- The maximum name length is 64 characters.

-- Avoid using names of built-in MySQL functions.

-- The last part of "CREATE PROCEDURE" is a pair of parentheses. "()" holds the parameter(s) list as there are no parameters in this procedure, the parameter list is empty.

- Next part is SELECT * FROM JOBS; $$ which is the last statement of the procedure body. Here the semicolon (;) is optional as $$ is a real statement-ender.

Call a procedure

The CALL statement is used to invoke a procedure that is stored in a DATABASE. Here is the 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. Let execute the procedure.

```
mysql> CALL  job_data$$
+------------+---------------------------------+------------+------------+
| JOB_ID     | JOB_TITLE                       | MIN_SALARY | MAX_SALARY |
+------------+---------------------------------+------------+------------+
| AD_PRES    | President                       |      20000 |      40000 |
| AD_VP      | Administration Vice President   |      15000 |      30000 |
| AD_ASST    | Administration Assistant        |       3000 |       6000 |
| FI_MGR     | Finance Manager                 |       8200 |      16000 |
| FI_ACCOUNT | Accountant                      |       4200 |       9000 |
| AC_MGR     | Accounting Manager              |       8200 |      16000 |
| AC_ACCOUNT | Public Accountant               |       4200 |       9000 |
| SA_MAN     | Sales Manager                   |      10000 |      20000 |
| SA_REP     | Sales Representative            |       6000 |      12000 |
| PU_MAN     | Purchasing Manager              |       8000 |      15000 |
| PU_CLERK   | Purchasing Clerk                |       2500 |       5500 |
| ST_MAN     | Stock Manager                   |       5500 |       8500 |
| ST_CLERK   | Stock Clerk                     |       2000 |       5000 |
| SH_CLERK   | Shipping Clerk                  |       2500 |       5500 |
| IT_PROG    | Programmer                      |       4000 |      10000 |
| MK_MAN     | Marketing Manager               |       9000 |      15000 |
| MK_REP     | Marketing Representative        |       4000 |       9000 |
| HR_REP     | Human Resources Representative  |       4000 |       9000 |
| PR_REP     | Public Relations Representative |       4500 |      10500 |
+------------+---------------------------------+------------+------------+
19 rows in set (0.00 sec)Query OK, 0 rows affected (0.15 sec)
```

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. Here is the syntax :

SHOW CREATE PROCEDURE proc_name

Let execute the above and see the output :

mysql> SHOW CREATE PROCEDURE job_data$$

MySQL : Characteristics Clauses

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 :

```
mysql> CREATE PROCEDURE job_data()
> SELECT * FROM JOBS; $$
Query OK, 0 rows affected (0.00 sec)
```

is the same as :

```
mysql> CREATE PROCEDURE new_job_data()
-> COMMENT ''
-> LANGUAGE SQL
-> NOT DETERMINISTIC
-> CONTAINS SQL
-> SQL SECURITY DEFINER
-> SELECT * FROM JOBS;
-> $$
Query OK, 0 rows affected (0.26 sec)
```

In the next section we will discuss on parameters

Before going to MySQL parameters let discuss some MySQL compound statements :

MySQL : 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. As of version 5.6 MySQL have following compound statements :

- BEGIN ... END Compound-Statement
- Statement Label
- DECLARE
- Variables in Stored Programs
- Flow Control Statements
- Cursors
- Condition Handling

In this section we will discuss the first four statements to cover the parameters part of CREATE PROCEDURE statement.

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). Here is the 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.

begin_label, end_label : See the following section.

Label Statement

Labels are permitted for BEGIN ... END blocks and for the LOOP, REPEAT, and WHILE statements. Here is the syntax :

[*begin_label*:]
BEGIN
[*statement_list*]
END [*end_label*]
[*begin_label*:]
LOOP
statement_list
END LOOP
[*end_label*]
[*begin_label*:]
REPEAT
statement_list
UNTIL *search_condition*

END
REPEAT [*end_label*]
[*begin_label*:]
WHILE *search_condition*
DO
statement_list
END WHILE
[*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.

Example: 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.

```
DELIMITER $$
CREATE PROCEDURE my_procedure_Local_Variables()
BEGIN   /* declare local variables */
DECLARE a INT DEFAULT 10;
DECLARE b, c INT;    /* using the local variables */
SET a = a + 100;
SET b = 2;
SET c = a + b;
BEGIN      /* local variable in nested block */
DECLARE c INT;
SET c = 5;
/* local variable c takes precedence over the one of the
same name declared in the enclosing block. */
SELECT a, b, c;
END;
SELECT a, b, c;
END$$
```

Now execute the procedure :

```
mysql> CALL my_procedure_Local_Variables();
+------+------+------+
| a    | b    | c    |
+------+------+------+
|  110 |    2 |    5 |
+------+------+------+
1 row in set (0.00 sec)

+------+------+------+
| a    | b    | c    |
+------+------+------+
|  110 |    2 |  112 |
+------+------+------+
1 row in set (0.01 sec)
Query OK, 0 rows affected (0.03 sec)
```

Example : 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 :

```
DELIMITER $$
```

```
CREATE PROCEDURE my_procedure_User_Variables()
BEGIN
SET @x = 15;
SET @y = 10;
SELECT @x, @y, @x-@y;
END$$
```

Now execute the procedure :

```
mysql> CALL my_procedure_User_Variables() ;
+------+------+-------+
| @x   | @y   | @x-@y |
+------+------+-------+
|   15 |   10 |     5 |
+------+------+-------+
1 row in set (0.04 sec)
Query OK, 0 rows affected (0.05 sec)
```

MySQL : Procedure Parameters

Here is the parameter part of CREATE PROCEDURE 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 () ...

2. CREATE PROCEDURE sp_name ([IN] param_name type)...

3. CREATE PROCEDURE sp_name ([OUT] param_name type)...

4. CREATE PROCEDURE sp_name ([INOUT] param_name type)...

In the first example, the parameter list is empty.

In the second examp,le an 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.

In the third example, 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.

In the fourth example, 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.

MySQL Procedure : 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. Here is the procedure :

```
mysql> CREATE PROCEDURE my_proc_IN (IN var1 INT)
-> BEGIN
-> SELECT * FROM jobs LIMIT var1;
->                                                                                                    END$$
Query OK, 0 rows affected (0.00 sec)
```

To execute the first 2 rows from the 'jobs' table execute the following command :

```
mysql> CALL my_proc_in(2)$$
+---------+------------------------------+------------+------------+
| JOB_ID  | JOB_TITLE                    | MIN_SALARY | MAX_SALARY |
+---------+------------------------------+------------+------------+
| AD_PRES | President                    |      20000 |      40000 |
| AD_VP   | Administration Vice President |      15000 |      30000 |
+---------+------------------------------+------------+------------+
2 rows in set (0.00 sec)Query OK, 0 rows affected (0.03 sec)
```

Now execute the first 5 rows from the 'jobs' table :

```
mysql>
CALL my_proc_in(5)$$
+------------+------------------------------+------------+------------+
| JOB_ID     | JOB_TITLE                    | MIN_SALARY | MAX_SALARY |
+------------+------------------------------+------------+------------+
| AD_PRES    | President                    |      20000 |      40000 |
| AD_VP      | Administration Vice President |      15000 |      30000 |
| AD_ASST    | Administration Assistant     |       3000 |       6000 |
| FI_MGR     | Finance Manager              |       8200 |      16000 |
| FI_ACCOUNT | Accountant                   |       4200 |       9000 |
```

+------------+-----------------------------+------------+------------+
5 rows in set (0.00 sec)Query OK, 0 rows affected (0.05 sec)

MySQL Procedure : 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.

```
mysql> CREATE PROCEDURE my_proc_OUT (OUT highest_salary INT)
-> BEGIN
-> SELECT MAX(MAX_SALARY) INTO highest_salary FROM JOBS;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

In the body of the procedure, the parameter will get the highest salary from MAX_SALARY column. After calling the procedure the word OUT tells the DBMS that the value goes out from the procedure. Here highest_salary is the name of the output parameter and we have passed its value to a session variable named @M, in the CALL statement.

```
mysql> CALL my_proc_OUT(@M)$$
Query OK, 1 row affected (0.03 sec)

mysql< SELECT @M$$+-------+
| @M    |
+-------+
| 40000 |
+-------+
1 row in set (0.00 sec)
```

MySQL Procedure : Parameter INOUT example

The following example shows a simple stored procedure that uses an INOUT parameter and an IN parameter. 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. Here is the code and output of the procedure :

mysql> CALL my_proc_OUT(@M)$$Query OK, 1 row affected (0.03 sec)mysql> CREATE PROCEDURE my_proc_INOUT (INOUT mfgender INT, IN emp_gender CHAR(1))

```
-> BEGIN
-> SELECT COUNT(gender) INTO mfgender FROM user_details WHERE gender = emp_gender;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

Now check the number of **male** and **female** users of the said tables :

```
mysql> CALL my_proc_INOUT(@C,'M')$$
Query OK, 1 row affected (0.02 sec)

mysql> SELECT @C$$
+------+
```

```
| @C   |
+------+
|    3 |
+------+
1 row in set (0.00 sec)

mysql> CALL my_proc_INOUT(@C,'F')$$
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @C$$
+------+
| @C   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

MySQL : 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.

MySQL : 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

- If *the condition* evaluates to true, the corresponding THEN or ELSEIF clause *statements(s)* executes.
- If no *condition* matches, the ELSE clause *statement(s)* executes.
- Each statement(s) consists of one or more SQL statements; an empty statement(s) is not permitted.

Example:

In the following example, we pass user_id through IN parameter to get the user name. Within the procedure, we have used IF ELSEIF and ELSE statement to get user name against multiple user id. The user name will be stored into INOUT parameter user_name.

```
CREATE DEFINER=`root`@`127.0.0.1`
PROCEDURE `GetUserName`(INOUT user_name varchar(16),
IN user_id varchar(16))
BEGIN
DECLARE uname varchar(16);
SELECT name INTO uname
FROM user
WHERE userid = user_id;
IF user_id = "scott123"
THEN
SET user_name = "Scott";
ELSEIF user_id = "ferp6734"
THEN
SET user_name = "Palash";
ELSEIF user_id = "diana094"
THEN
SET user_name = "Diana";
END IF;
END
```

Execute the procedure:

```
mysql> CALL GetUserName(@A,'scott123')$$
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @A;
   -> $$
+-------+
| @A    |
+-------+
| Scott |
+-------+
1 row in set (0.00 sec)
```

MySQL : 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
```

Explanation: First syntax

- case_value is an expression.
- This value is compared to the when_value expression in each WHEN clause until one of them is equal.
- When an equal when_value is found, the corresponding THEN clause statement_list executes.
- If no when_value is equal, the ELSE clause statement_list executes, if there is one.

Explanation: Second syntax

- Each WHEN clause search_condition expression is evaluated until one is true, at which point its corresponding THEN clause statement_list executes.
- If no search_condition is equal, the ELSE clause statement_list executes, if there is one.
- Each statement_list consists of one or more SQL statements; an empty statement_list is not permitted.

Example:

We have table called 'jobs' with following records :

```
+------------+---------------------------------+------------+------------+
| JOB_ID     | JOB_TITLE                       | MIN_SALARY | MAX_SALARY |
+------------+---------------------------------+------------+------------+
| AD_PRES    | President                       |      20000 |      40000 |
| AD_VP      | Administration Vice President   |      15000 |      30000 |
| AD_ASST    | Administration Assistant        |       3000 |       6000 |
| FI_MGR     | Finance Manager                 |       8200 |      16000 |
| FI_ACCOUNT | Accountant                      |       4200 |       9000 |
| AC_MGR     | Accounting Manager              |       8200 |      16000 |
| AC_ACCOUNT | Public Accountant               |       4200 |       9000 |
| SA_MAN     | Sales Manager                   |      10000 |      20000 |
| SA_REP     | Sales Representative            |       6000 |      12000 |
| PU_MAN     | Purchasing Manager              |       8000 |      15000 |
| PU_CLERK   | Purchasing Clerk                |       2500 |       5500 |
| ST_MAN     | Stock Manager                   |       5500 |       8500 |
| ST_CLERK   | Stock Clerk                     |       2000 |       5000 |
| SH_CLERK   | Shipping Clerk                  |       2500 |       5500 |
| IT_PROG    | Programmer                      |       4000 |      10000 |
| MK_MAN     | Marketing Manager               |       9000 |      15000 |
| MK_REP     | Marketing Representative        |       4000 |       9000 |
| HR_REP     | Human Resources Representative  |       4000 |       9000 |
| PR_REP     | Public Relations Representative |       4500 |      10500 |
+------------+---------------------------------+------------+------------+
19 rows in set (0.03 sec)
```

Now we want to count the number of employees with following conditions :

-	MIN_SALARY	>	10000
-	MIN_SALARY	<	10000

- MIN_SALARY = 10000

Here is the procedure (the procedure is written into MySQL workbench 5.2 CE) :

```
DELIMITER $$
CREATE PROCEDURE `hr`.`my_proc_CASE`
(INOUT no_employees INT, IN salary INT)
BEGIN
CASE
WHEN (salary>10000)
THEN (SELECT COUNT(job_id) INTO no_employees
FROM jobs
WHERE min_salary>10000);
WHEN (salary<10000)
THEN (SELECT COUNT(job_id) INTO no_employees
FROM jobs
WHERE min_salary<10000);
ELSE (SELECT COUNT(job_id) INTO no_employees
FROM jobs WHERE min_salary=10000);
END CASE;
END$$
```

In the above procedure, we pass the salary (amount) variable through IN parameter. Within the procedure, there is CASE statement along with two WHEN and an ELSE which will test the condition and return the count value in no_employees. Let execute the procedure in MySQL command prompt :

Number of employees whose salary greater than 10000 :

```
mysql> CALL my_proc_CASE(@C,10001);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @C;
+------+
| @C   |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

Number of employees whose salary less than 10000 :

```
mysql> CALL my_proc_CASE(@C,9999);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @C;
+------+
| @C   |
```

```
+------+
|   16 |
+------+
1 row in set (0.00 sec)
```

Number of employees whose salary equal to 10000 :

```
mysql> CALL my_proc_CASE(@C,10000);
Query OK, 1 row affected (0.00 sec)
mysql> SELECT @C;
+------+
| @C   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

MySQL: ITERATE Statement

ITERATE means "start the loop again". ITERATE can appear only within LOOP, REPEAT, and WHILE statements. Here is the syntax :

ITERATE label

MySQL: 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). Here is the syntax :

LEAVE label

MySQL : LOOP Statement

LOOP is used to create repeated execution of the statement list. Here is the 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.

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

Now execute the procedure :

```
mysql> CALL my_proc_LOOP(3);
Query OK, 1 row affected, 1 warning (0.19 sec)

mysql> select * from number;
+--------------+
| rnumber      |
+--------------+
| 0.1228974146 |
| 0.2705919913 |
| 0.9842677433 |
+--------------+
3 rows in set (0.00 sec)
```

MySQL: 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:

Even numbers are numbers that can be divided evenly by 2. In the following procedure an user passes a number through IN parameter and make a sum of even numbers between 1 and that particular number.

```
DELIMITER $$
CREATE PROCEDURE my_proc_REPEAT (IN n INT)
BEGI
NSET @sum = 0;
SET @x = 1;
REPEAT
IF mod(@x, 2) = 0
THEN
SET @sum = @sum + @x;
END IF;
SET @x = @x + 1;
UNTIL @x > n
END REPEAT;
END $$
```

Now execute the procedure:

```
mysql> call my_proc_REPEAT(5);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @sum;
+------+
| @sum |
+------+
|    6 |
+------+
1 row in set (0.00 sec)

mysql> call my_proc_REPEAT(10);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @sum;
+------+
| @sum |
+------+
|   30 |
+------+
1 row in set (0.00 sec)
```

MySQL: 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. Here is the 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.

MySQL : 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 (;). Here is the syntax:

[begin_label:] WHILE search_condition DO

statement_list

END WHILE [end_label]

A WHILE statement can be labeled.

Example:

Odd numbers are numbers that cannot be divided exactly by 2. 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.

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

Now execute the procedure:

```
mysql> CALL my_proc_WHILE(5);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @sum;
+------+
| @sum |
+------+
|    3 |
+------+
1 row in set (0.00 sec)

mysql> CALL my_proc_WHILE(10);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @sum;
+------+
| @sum |
+------+
|   25 |
+------+
1 row in set (0.00 sec)
mysql> CALL my_proc_WHILE(3);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @sum;
+------+
| @sum |
+------+
|    4 |
+------+
1 row in set (0.00 sec)
```

MySQL: 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. Here is the 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. In our previous procedure "my_proc_WHILE" the comment section was empty. To input new comment or modify the previous comment use the following command :

```
mysql> ALTER PROCEDURE my_proc_WHILE
COMMENT 'Modify Comment';
>Query OK, 0 rows affected (0.20 sec)
```

You can check the result through SHOW CREATE PROCEDURE command which we have discussed earlier.

MySQL: 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. (If the automatic_sp_privileges system variable is enabled, that privilege and EXECUTE are granted automatically to the routine creator when the routine is created and dropped from the creator when the routine is dropped

```
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. Here is an example:

```
mysql> DROP PROCEDURE new_procedure;
Query OK, 0 rows affected (0.05 sec)
```

You can check the result through SHOW CREATE PROCEDURE command which we have discussed earlier.

MySQL: 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

To use cursors in MySQL procedures, you need to do the following :
- 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.

```
DELIMITER $$
CREATE PROCEDURE my_procedure_cursors(INOUT return_val INT)
BEGIN
DECLARE a,b INT;
DECLARE cur_1 CURSOR FOR
SELECT max_salary FROM jobs;
DECLARE CONTINUE HANDLER FOR NOT FOUNDSET b = 1;
OPEN cur_1;REPEATFETCH cur_1 INTO a;
UNTIL b = 1END REPEAT;
CLOSE cur_1;
SET return_val = a;
END;
$$
```

Now execute the procedure:

```
mysql>
CALL my_procedure_cursors(@R);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @R;
+-------+
| @R    |
+-------+
| 10500 |
+-------+
1 row in set (0.00 sec)
```

We will provide more examples on cursors soon.

Access Control for Stored Programs

Stored programs and views are defined prior to use and, when referenced, execute within a security context that determines their privileges. These privileges are controlled by their DEFINER attribute, and, if there is one, their SQL SECURITY characteristic.

All stored programs (procedures, functions, and triggers) and views can have a DEFINER attribute that names a MySQL account. If the DEFINER attribute is omitted from a stored program or view definition, the default account is the user who creates the object.

MySQL uses the following rules to control which accounts a user can specify in an object DEFINER attribute :

- You can specify a DEFINER value other than your own account only if you have the SUPER privilege.

- If you do not have the SUPER privilege, the only legal user value is your own account, either specified literally or by using CURRENT_USER. You cannot set the definer to some other account.
- For a stored routine or view, use SQL SECURITY INVOKER in the object definition when possible so that it can be used only by users with permissions appropriate for the operations performed by the object.
- If you create definer-context stored programs or views while using an account that has the SUPER privilege, specify an explicit DEFINER attribute that names an account possessing only the privileges required for the operations performed by the object. Specify a highly privileged DEFINER account only when absolutely necessary.
- Administrators can prevent users from specifying highly privileged DEFINER accounts by not granting them the SUPER privilege.
- Definer-context objects should be written keeping in mind that they may be able to access data for which the invoking user has no privileges. In some cases, you can prevent reference to these objects by not granting unauthorized users particular privileges:
- A stored procedure or function cannot be referenced by a user who does not have the EXECUTE privilege for it.
- A view cannot be referenced by a user who does not have the appropriate privilege for it (SELECT to select from it, INSERT to insert into it, and so forth).