**Stored Procedure**

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, 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.
* 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:**

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.

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

**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 $$ ;

Example:

1. 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)

SHOW CREATE PROCEDURE job\_data$$

**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[()]

**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)

**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*]

**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$$

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$$

Copy

**Now execute the procedure**

mysql> CALL my\_procedure\_User\_Variables() ;

+------+------+-------+

| @x | @y | @x-@y |

+------+------+-------+

| 15 | 10 | 5 |

+------+------+-------+

**Procedure Parameters:**

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.

**Example:**

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.

**EX**: IN parameter

create procedure second\_max\_salaray(in var int)

begin

select max\_salary

from jobs

where max\_salary <

(

select max(max\_salary) from jobs

) limit var;

end;$$

call second\_max\_salaray(1);

1. Out parameter:

delimiter $$

create procedure max\_salary(out heighest\_salary int)

begin

select max(max\_salary) into heighest\_salary

from jobs;

end;$$

call max\_salary(@salary);$$

select @salary; $$

1. INOUT parameter :

delimiter $$

create procedure employee\_count\_in\_Dept(inout no\_of\_employee int,in dept\_id int)

begin

select count(\*) into no\_of\_employee

from employees

where DEPARTMENT\_ID = dept\_id;

end;$$

call employee\_count\_in\_Dept(@count,'50');

select @count;

**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.

* 1. **If Statement:**

/\*Procedure with if else condition\*/

delimiter $$

create procedure GetUserName(inout emp\_name varchar(16),in emp\_id int)

begin

DECLARE uname varchar(16);

select FIRST\_NAME into uname

from employees

where employee\_id = emp\_id;

if emp\_id = 100

then

set emp\_name = 'Steven King';

elseif emp\_id = 103

then

set emp\_name = 'Lex';

else

set emp\_id = 'no name found';

end if;

end;$$

call GetUserName(@name,103);

select @name;

* 1. **Case:**

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

DELIMITER $$

CREATE PROCEDURE `hr`.`CaseProcedure` (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 CaseProcedure(@count,1000);

select @count

mysql> SELECT @C;

+------+

| @C |

+------+

| 16 |

+------+

1 row in set (0.00 sec)

**ITERATE Statement:**

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

LEAVE label

**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]

Example:

create table number(double int);

DELIMITER $$

CREATE PROCEDURE `LoopProcedure` (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 LoopProcedure(10);

select \* from number;

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)

**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]

**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.

/\* Repeate Statement \*/

delimiter $$

create procedure RepeateProcedure(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 n < @x

end repeat;

end;$$

call RepeateProcedure(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. 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.

**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]

**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.

mysql> CALL my\_proc\_WHILE(5);

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @sum;

+------+

| @sum |

+------+

| 3 |

+------+

**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)

**DROP PROCEDURE:**

This statement is used to drop a stored procedure or function.

DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp\_name

EX:

mysql> DROP PROCEDURE new\_procedure;

Query OK, 0 rows affected (0.05 sec)

**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.

/\*Cursor example\*/

delimiter $$

create procedure CursorProcedure(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 CursorProcedure(@salary);

select @salary;