MySQL Stored Procedure

A procedure (often called a stored procedure) is a **collection of pre-compiled SQL statements** stored inside the database. It is a subroutine or a subprogram in the regular computing language. **A procedure always contains a name, parameter lists, and SQL statements**. We can invoke the procedures by using triggers, other procedures and applications such as [Java](https://www.javatpoint.com/java-tutorial), [Python](https://www.javatpoint.com/python-tutorial), [PHP](https://www.javatpoint.com/php-tutorial), etc. It was first introduced in MySQL **version 5**. Presently, it can be supported by almost all relational database systems.

If we consider the enterprise application, we always need to perform specific tasks such as database cleanup, processing payroll, and many more on the database regularly. Such tasks involve multiple [SQL](https://www.javatpoint.com/sql-tutorial) statements for executing each task. This process might easy if we group these tasks into a single task. We can fulfill this requirement in [MySQL](https://www.javatpoint.com/mysql-tutorial) by creating a stored procedure in our database.

A procedure is called a **recursive stored procedure** when it calls itself. Most database systems support recursive stored procedures. But, it is not supported well in MySQL.

Stored Procedure Features

Stored Procedure increases the performance of the applications. Once stored procedures are created, they are compiled and stored in the database.

* Stored procedure reduces the traffic between application and database server. Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements.
* Stored procedures are reusable and transparent to any applications.
* A procedure is always secure. The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.

How to create a procedure?

The following syntax is used for creating a stored procedure in MySQL. It can return one or more value through parameters or sometimes may not return at all. By default, a procedure is associated with our current database. But we can also create it into another database from the current database by specifying the name as **database\_name.procedure\_name**. See the complete syntax:

DELIMITER &&

**CREATE** **PROCEDURE** procedure\_name [[IN | **OUT** | INOUT] parameter\_name datatype [, parameter datatype]) ]

**BEGIN**

    Declaration\_section

    Executable\_section

**END** &&

DELIMITER ;

Parameter Explanations

The procedure syntax has the following parameters:

|  |  |
| --- | --- |
| **Parameter Name** | **Descriptions** |
| procedure\_name | It represents the name of the stored procedure. |
| parameter | It represents the number of parameters. It can be one or more than one. |
| Declaration\_section | It represents the declarations of all variables. |
| Executable\_section | It represents the code for the function execution. |

**MySQL procedure parameter has one of three modes:**

**IN parameter**

It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

**OUT parameters**

It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

**INOUT parameters**

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

How to call a stored procedure?

We can use the **CALL statement** to call a stored procedure. This statement returns the values to its caller through its parameters (IN, OUT, or INOUT). The following syntax is used to call the stored procedure in MySQL:

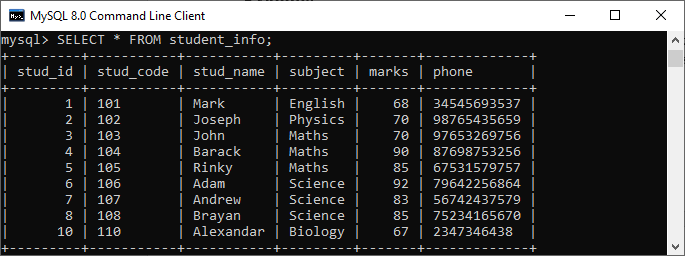
1. CALL procedure\_name ( parameter(s))

Example

Let us understand how to create a procedure in MySQL through example. First, we need to select a database that will store the newly created procedure. We can select the database using the below statement:

1. mysql> USE database\_name;

Suppose this database has a table named **student\_info** that contains the following data:



CREATE TABLE student\_info (

stu\_id INT,

stud\_code VARCHAR(10),

name VARCHAR(50),

subject VARCHAR(50),

marks INT,

phone BIGINT

);

INSERT INTO student\_info (stu\_id, stud\_code, name, subject, marks, phone) VALUES

(1, '101', 'Mark', 'English', 68, 3445465937),

(2, '102', 'Joseph', 'Physics', 70, 9876543959),

(3, '103', 'John Doe', 'Maths', 85, 9765326565),

(4, '104', 'Barack', 'Maths', 90, 8765975757),

(5, '105', 'Rincky', 'Science', 98, 6753819256),

(6, '106', 'Adam Newman', 'Science', 92, 7642244874),

(7, '107', 'Alexander Bellamy', 'Social Science', 85, 5904653664),

(8, '108', 'Brayan Alexander', 'Biology', 67, 7237416470),

(10, '110', 'Alexandar', 'Biology', 67, 2347346489);

Procedure without Parameter

Suppose we want **to display all records of this table whose marks are greater than 70** and count all the table rows. The following code creates a procedure named **get\_merit\_students**:

DELIMITER &&

**CREATE** **PROCEDURE** get\_merit\_student ()

**BEGIN**

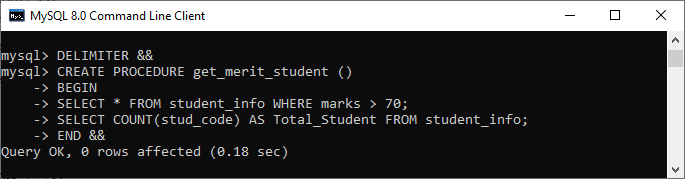
**SELECT** \* **FROM** student\_info **WHERE** marks > 70;

**SELECT** COUNT(stud\_code) **AS** Total\_Student **FROM** student\_info;

**END** &&

DELIMITER ;

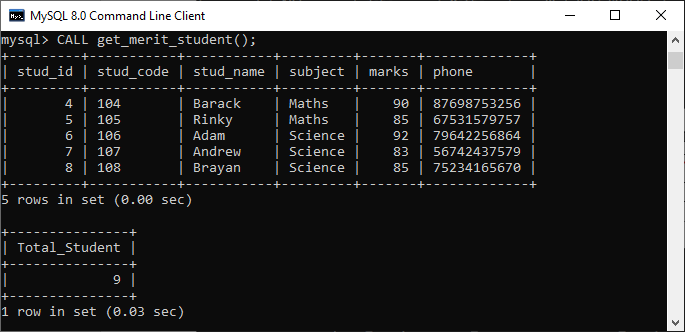
If this code executed successfully, we would get the below output:



Let us call the procedure to verify the output:

1. mysql> CALL get\_merit\_student();

It will give the output as follows:



Procedures with IN Parameter

In this procedure, we have used the IN parameter as **'var1**' of integer type to accept a number from users. Its body part fetches the records from the table using a [**SELECT** statement](https://www.javatpoint.com/mysql-select) and returns only those rows that will be supplied by the user. It also returns the total number of rows of the specified table. See the procedure code:

DELIMITER &&

**CREATE** **PROCEDURE** get\_student (IN var1 **INT**)

**BEGIN**

**SELECT** \* **FROM** student\_info LIMIT var1;

**SELECT** COUNT(stud\_code) **AS** Total\_Student **FROM** student\_info;

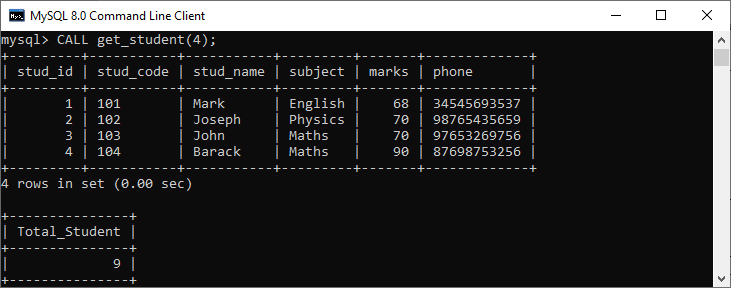
**END** &&

DELIMITER ;

After successful execution, we can call the procedure as follows:

1. mysql> CALL get\_student(4);

We will get the below output:



Procedures with OUT Parameter

In this procedure, we have used the OUT parameter as the **'highestmark'** of integer type. Its body part fetches the maximum marks from the table using a **MAX() function**. See the procedure code:

DELIMITER &&

**CREATE** **PROCEDURE** display\_max\_mark (**OUT** highestmark **INT**)

**BEGIN**

**SELECT** **MAX**(marks) **INTO** highestmark **FROM** student\_info;

**SELECT** **AVG**(marks) **INTO** highestmark **FROM** student\_info1;

**END** &&

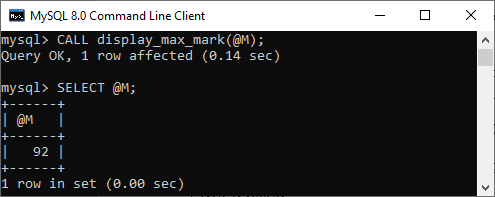
DELIMITER ;

This procedure's parameter will get the highest marks from the **student\_info** table. When we call the procedure, the OUT parameter tells the database systems that its value goes out from the procedures. Now, we will pass its value to a session variable **@M** in the CALL statement as follows:

mysql> CALL display\_max\_mark(@M);

mysql> **SELECT** @M;

Here is the output:



Procedures with INOUT Parameter

In this procedure, we have used the INOUT parameter as **'var1'** of integer type. Its body part first fetches the marks from the table with the specified **id** and then stores it into the same variable var1. The var1 first acts as the IN parameter and then OUT parameter. Therefore, we can call it the INOUT parameter mode. See the procedure code:

DELIMITER &&

**CREATE** **PROCEDURE** display\_marks (INOUT var1 **INT**)

**BEGIN**

**SELECT** marks **INTO** var1 **FROM** student\_info **WHERE** stu\_id = var1;

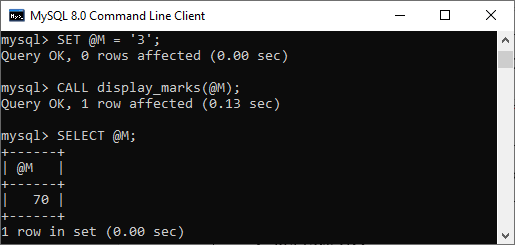
**END** &&

DELIMITER ;

After successful execution, we can call the procedure as follows:

1. mysql> **SET** @M1 = '3';
2. mysql> CALL display\_marks(@M1);
3. mysql> **SELECT** @M1;

We will get the below output:



How to show or list stored procedures in MySQL?

When we have several procedures in the MySQL server, it is very important to list all procedures. It is because sometimes the procedure names are the same in many databases. In that case, this query is very useful. We can list all procedure stored on the current MySQL server as follows:

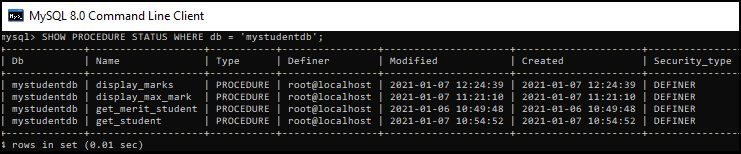
1. SHOW **PROCEDURE** STATUS [LIKE 'pattern' | **WHERE** search\_condition]

**This statement displays all stored procedure names, including their characteristics**. If we want to display procedures in a particular database, we need to use the [**WHERE** clause](https://www.javatpoint.com/mysql-where). In case we want to list stored procedures with a specific word, we need to use the [**LIKE** clause](https://www.javatpoint.com/mysql-like).

We can list all stored procedure in the MySQL **mystudentsb** database using the below statement:

1. mysql> SHOW **PROCEDURE** STATUS **WHERE** db = 'practicedb';

It will give the below output where we can see that the mystudentdb database contains **four stored procedures**:



How to delete/drop stored procedures in MySQL?

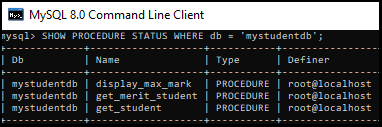
MySQL also allows a command to drop the procedure. When the procedure is dropped, it is removed from the database server also. The following statement is used to drop a stored procedure in MySQL:

1. **DROP** **PROCEDURE** [ IF EXISTS ] procedure\_name;

Suppose we want to remove the procedure named **display\_marks** from the mystudentdb database. We can do this by first selecting the database and then use the syntax as follows to remove the procedure:

1. mysql> **DROP** **PROCEDURE** display\_marks;

We can verify it by listing the procedure in the specified database using the **SHOW PROCEDURE STATUS** command. See the below output:



**NEXT PROBLEM**

**Example:** Consider two tables author and book:

-- Create the author table

CREATE TABLE author (

author\_id INTEGER PRIMARY KEY,

authorName VARCHAR(50), -- Increased length for names

email VARCHAR(50), -- Increased length for email

gender VARCHAR(6)

);

-- Create the book table

CREATE TABLE book (

BookId INTEGER NOT NULL UNIQUE,

ISBN VARCHAR(10) PRIMARY KEY, -- Changed to VARCHAR to handle leading zeros

book\_name VARCHAR(100) NOT NULL, -- Increased length for book names

author INTEGER,

ed\_num INTEGER,

price INTEGER,

pages INTEGER,

FOREIGN KEY (author) REFERENCES author (author\_id) ON DELETE CASCADE

);

-- Inserting values into the author table

INSERT INTO author (author\_id, authorName, email, gender) VALUES

(1, 'Kraig Muller', 'Wordnewton@gmail.com', 'Male'),

(2, 'Karrie Nicolette', 'karrie23@gmail.com', 'Female');

-- Inserting values into the book table

INSERT INTO book (BookId, ISBN, book\_name, author, ed\_num, price, pages) VALUES

(1, '001', 'Glimpses of the past', 1, 1, 650, 396),

(2, '002', 'Beyond The Horizons of Venus', 1, 1, 650, 396),

(3, '003', 'Ultrasonic Aquaculture', 2, 1, 799, 500),

(4, '004', 'Cryogenic Engines', 2, 1, 499, 330);

Procedure (with no parameters) to display all the books:

delimiter //

create procedure display\_book()

begin

select \*from book;

end //

call display\_book(); //

delimiter ;

1. **Procedure with IN parameter:** An IN parameter is used to take a parameter as input such as an attribute. When we define an IN parameter in a 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 if the value of the IN parameter is changed inside the procedure, its original value is retained after the procedure ends (like pass by value). In other words, the procedure only works on the copy of the IN parameter.

**Example:** Procedure to update price of a book taking ISBN of the book and its new price as input: (considering the tables above)

DELIMITER //

CREATE PROCEDURE update\_price (IN temp\_ISBN VARCHAR(10), IN new\_price INTEGER)

BEGIN

UPDATE book SET price = new\_price WHERE ISBN = temp\_ISBN;

END //

DELIMITER ;

-- Call the procedure

CALL update\_price('001', 600);

SELECT \* FROM BOOK;

1. **Procedure with OUT parameter:** An OUT parameter is used to pass a parameter as output or display like the select operator, but implicitly (through a set value). The value of an OUT parameter can be changed inside the procedure and its new value is passed back to the calling program. A procedure cannot access the initial value of the OUT parameter when it starts.

**Example:** Procedure to display the highest price among all the books with an output parameter:

delimiter //

create procedure disp\_max(OUT highestprice integer)

begin

select max(price) into highestprice from book;

end; //

call disp\_max(@M); //

select @M; //

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

**Example:** Procedure to take gender type input (‘Male’/’Female’ here) with an in-out parameter which reflects the number of authors falling in that gender category/type:

delimiter //

create procedure disp\_gender(INOUT mfgender integer, IN emp\_gender varchar(6))

begin

select COUNT(gender) INTO mfgender FROM author where gender = emp\_gender;

end; //

delimiter ;

call disp\_gender(@M, "Male");

select @M;

call disp\_gender(@F, "Female");

select @F;

Invoked the Procedures using Triggers:

**-- Step 1: Create the price\_log table**

CREATE TABLE price\_log (

log\_id INT AUTO\_INCREMENT PRIMARY KEY,

ISBN VARCHAR(10),

old\_price INT,

new\_price INT,

change\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Step 2: Set the delimiter

DELIMITER //

**-- Step 3: Create the stored procedure**

CREATE PROCEDURE log\_price\_change(

IN p\_ISBN VARCHAR(10),

IN p\_old\_price INT,

IN p\_new\_price INT

)

BEGIN

INSERT INTO price\_log (ISBN, old\_price, new\_price)

VALUES (p\_ISBN, p\_old\_price, p\_new\_price);

END //

-- Step 4: Reset the delimiter

DELIMITER ;

**-- Step 5: Create the trigger**

DELIMITER //

CREATE TRIGGER before\_price\_update

BEFORE UPDATE ON book

FOR EACH ROW

BEGIN

CALL log\_price\_change(OLD.ISBN, OLD.price, NEW.price);

END //

-- Step 6: Reset the delimiter

DELIMITER ;

**Testing the Trigger**

To test if the trigger and procedure work correctly, you can update the price in the book table and then check the price\_log table.

-- Update the price of a book to test the trigger

UPDATE book SET price = 700 WHERE ISBN = '001';

-- Check the price\_log table to see if the change was logged

SELECT \* FROM price\_log;

Stored procedures are a powerful feature in SQL that encapsulate a set of SQL statements and logic to be executed. Here are some advanced topics and features related to stored procedures that you might find useful:

1. **Error Handling**:
   * Use DECLARE to declare variables and handlers for error conditions.
   * Use SIGNAL and RESIGNAL statements to raise custom errors.

DELIMITER //

CREATE PROCEDURE safe\_division(IN dividend INT, IN divisor INT, OUT result FLOAT)

BEGIN

DECLARE division\_by\_zero CONDITION FOR SQLSTATE '45000';

DECLARE CONTINUE HANDLER FOR division\_by\_zero

SET result = NULL;

IF divisor = 0 THEN

SIGNAL division\_by\_zero SET MESSAGE\_TEXT = 'Division by zero is not allowed';

ELSE

SET result = dividend / divisor;

END IF;

END //

DELIMITER ;

SET @result = 0; -- Initialize the result variable

CALL safe\_division(10, 2, @result); -- Call the procedure with dividend = 10, divisor = 2

SELECT @result; -- Retrieve and display the result

SET @result = 0;

CALL safe\_division(10, 0, @result);

SELECT @result; -- This should show NULL due to division by zero

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

salary DECIMAL(10, 2),

grade INT

);

INSERT INTO employees (employee\_id, salary, grade) VALUES

(101, 50000.00, 1),

(102, 60000.00, 2),

(103, 70000.00, 3);

DELIMITER //

CREATE PROCEDURE calculate\_bonus(IN emp\_id INT, OUT bonus DECIMAL(10,2))

BEGIN

DECLARE emp\_salary DECIMAL(10,2);

DECLARE emp\_grade INT;

SELECT salary, grade INTO emp\_salary, emp\_grade FROM employees WHERE employee\_id = emp\_id;

CASE

WHEN emp\_grade = 1 THEN SET bonus = emp\_salary \* 0.1;

WHEN emp\_grade = 2 THEN SET bonus = emp\_salary \* 0.15;

WHEN emp\_grade = 3 THEN SET bonus = emp\_salary \* 0.2;

ELSE SET bonus = 0;

END CASE;

END //

DELIMITER ;

CALL calculate\_bonus(101, @emp\_bonus);

SELECT @emp\_bonus;