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, Python, PHP, 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 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 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 values 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:

```
CREATE PROCEDURE procedure_name [[IN | OUT | INOUT] parameter_name
datatype [, parameter datatype]) ]
BEGIN
   Declaration_section
   Executable_section
END
```

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:

```
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:

```
mysql> USE database_name;
```

Suppose this database has a table named **student_info** that contains the following data:

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_top_students**:

```
CREATE PROCEDURE get_top_student ()

BEGIN

   SELECT * FROM student_info WHERE marks > 70;

   SELECT COUNT(stud_code) AS Total_Student FROM student_info;

END
```

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

CALL get_top_student();

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 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:

```
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:

```
CALL get_student(4);
```

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:

```
CREATE PROCEDURE display_max_mark (OUT highestmark INT)

BEGIN

  SELECT MAX(marks) INTO highestmark FROM student_info;

END
```

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:

```
CALL display_max_mark(@M);

SELECT @M;
```

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:

```
CREATE PROCEDURE display_marks (INOUT var1 INT)

BEGIN

  SELECT marks INTO var1 FROM student_info WHERE stud_id = var1;

END
```

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

```
SET @M = '3';

CALL display_marks(@M);

SELECT @M;
```

We will get the below output:

Aim:

Create Student table(regno,name, English_mark,Malayalam_mark, physics_mark, chemistry_mark , maths_mark)
Insert 5 rows of table
Write a function which accepts the regno , calculate the total mark and return total mark.

Trigger

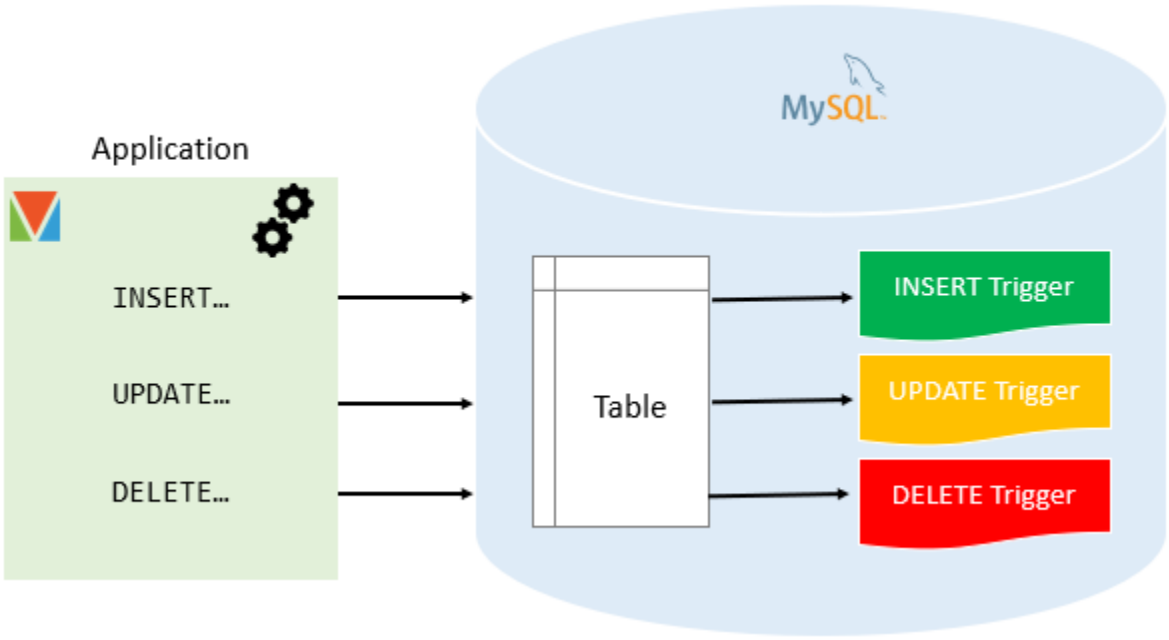
In MySQL, a trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table. For example, you can define a trigger that is invoked automatically before a new row is inserted into a table.

MySQL supports triggers that are invoked in response to the INSERT, UPDATE or DELETE event.

The SQL standard defines two types of triggers: row-level triggers and statement-level triggers.

- A row-level trigger is activated for each row that is inserted, updated, or deleted. For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected.
- A statement-level trigger is executed once for each transaction regardless of how many rows are inserted, updated, or deleted.

MySQL supports only row-level triggers. It doesn't support statement-level triggers.



Advantages of triggers

- Triggers provide another way to check the integrity of data.
- Triggers handle errors from the database layer.
- Triggers give an alternative way to run scheduled tasks. By using triggers, you don't have to wait for the scheduled events to run because the triggers are invoked automatically *before* or *after* a change is made to the data in a table.
- Triggers can be useful for auditing the data changes in tables.

Disadvantages of triggers

- Triggers can only provide extended validations, not all validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints.
- Triggers can be difficult to troubleshoot because they execute automatically in the database, which may not invisible to the client applications.
- Triggers may increase the overhead of the MySQL Server.

Managing MySQL triggers

- Create triggers – describe steps of how to create a trigger in MySQL.
- Drop triggers – show you how to drop a trigger.
- Create a BEFORE INSERT trigger – show you how to create a BEFORE INSERT trigger to maintain a summary table from another table.
- Create an AFTER INSERT trigger – describe how to create an AFTER INSERT trigger to insert data into a table after inserting data into another table.
- Create a BEFORE UPDATE trigger – learn how to create a BEFORE UPDATE trigger that validates data before it is updated to the table.
- Create an AFTER UPDATE trigger – show you how to create an AFTER UPDATE trigger to log the changes of data in a table.
- Create a BEFORE DELETE trigger – show how to create a BEFORE DELETE trigger.
- Create an AFTER DELETE trigger – describe how to create an AFTER DELETE trigger.
- Create multiple triggers for a table that have the same trigger event and time – MySQL 8.0 allows you to define multiple triggers for a table that have the same trigger event and time.
- Show triggers – list triggers in a database, table by specific patterns.

MySQL CREATE TRIGGER statement

The CREATE TRIGGER statement creates a new trigger. Here is the basic syntax of the CREATE TRIGGER statement:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

```
CREATE TABLE employee(
    name varchar(45) NOT NULL,
    occupation varchar(35) NOT NULL,
    working_date date,
    working_hours varchar(10)
);
INSERT INTO employee VALUES
```

```
('Anu', 'Scientist', '2020-10-04', 12),
('Binu', 'Engineer', '2020-10-04', 10),
('Cinu', 'Actor', '2020-10-04', 13),
('Dinu', 'Doctor', '2020-10-04', 14),
('Finu', 'Teacher', '2020-10-04', 12),
('Ginu', 'Business', '2020-10-04', 11)
```

Next, we will create a **BEFORE INSERT trigger**. This trigger is invoked automatically insert the **working_hours = 0** if someone tries to insert **working_hours < 0**.

```
Create Trigger before_insert_empworkinghours

BEFORE INSERT ON employee FOR EACH ROW

BEGIN

IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;

END IF;

END
```

To invoke this trigger

```
INSERT INTO employee VALUES

('Minu', 'Former', '2020-10-08', 14);


INSERT INTO employee VALUES

('Annu', 'Actor', '2020-10-012', -90);
```

In this output, we can see that on inserting the negative values into the working_hours column of the table will automatically fill the zero value by a trigger.

Question

employee (emp_no, emp_name, DOB, address, doj, designation, mobile_no, dept_no, salary).

salary(empno, old_sal, new_sal, rev_date)
personal_updations()
promotions()

1. Create a Trigger for the **employee** table that will store the updated salary into another table **SALARY** while updating salary.
2. Create a Trigger for the **employee** table that will store the updated mobile number into another table **personal_updations** while updating mobile number.
3. Create a Trigger for the **employee** table that will store the promotion details on updation.