
Stored procedures:

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedures are created to perform one or more DML operations on Database. It is nothing but the group of SQL statements that accepts some input in the form of parameters and performs some task and may or may not return a value.

> *Note: They might look similar to functions in other programming languages.*

`Syntax:`

```
DELIMITER //
  CREATE PROCEDURE Procedure_Name()
    BEGIN
       SELECT *  FROM table_name;
    END //
  DELIMITER ;
```

- The command `DELIMITER` is used to start defining a procedure, now before we start defining the `Procedure` the `DELIMITER` statement changes the standard delimiter which is a semicolon (;) to another which is again changed back to semicolon (;) by using `DELIMITER` at the end of our command. You may use your own delimiter example `$`.
- Following the `END` keyword, we use the delimiter // to indicate the end of the stored procedure. The last command (DELIMITER;) changes the delimiter back to the semicolon (;).
- The code between BEGIN and END is called the body of the stored procedure.

Let's create a procedure to retrieve names of all students from our School's database

```
DELIMITER //
use school //
  CREATE PROCEDURE stud_name()
    BEGIN
       SELECT *  FROM students;
    END //
DELIMITER ;
```

How to call a procedure:

```
CALL stud_name();
-- Use call procedure_name() to call the procedure.
-- It will get us names of all the students.
```

Benefits of using stored procedures:

The following list describes some benefits of using procedures.

1. Stored procedures help **increase the performance** of the applications. Once created, stored procedures are compiled and stored in the database.
2. Stored procedures help **reduce the traffic** between application and database serverbecause instead of sending multiple lengthy SQL statements, the application has to send only the name and parameters of the stored procedure.
3. Stored procedures are **reusable** and transparent to any applications.
4. Stored procedures are **secure**. The database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permissions on the underlying database tables.

DRAWBACKS OF USING STORED PROCEDURES:

1. If you use many stored procedures, the **memory usage** of every connection that is using those stored procedures will **increase** substantially. In addition, if you overuse a large number of logical operations inside stored procedures, the CPU usage will increase because the database server is not well-designed for logical operations.
    2. Stored procedure's constructs are **not designed** for developing **complex and flexible business logic**.
    3. It is **difficult** to **debug stored procedures**. Only a few database management systems allow you to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures.
    4. It is not easy to develop and maintain stored procedures. Developing and maintaining stored procedures are often required a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance phases.

> *Learn more about stored Procedures here: https://www.scaler.com/topics/stored-procedure-in-sql/ https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html https://www.sqlshack.com/learn-mysql-the-basics-of-mysql-stored-procedures/*

Triggers:

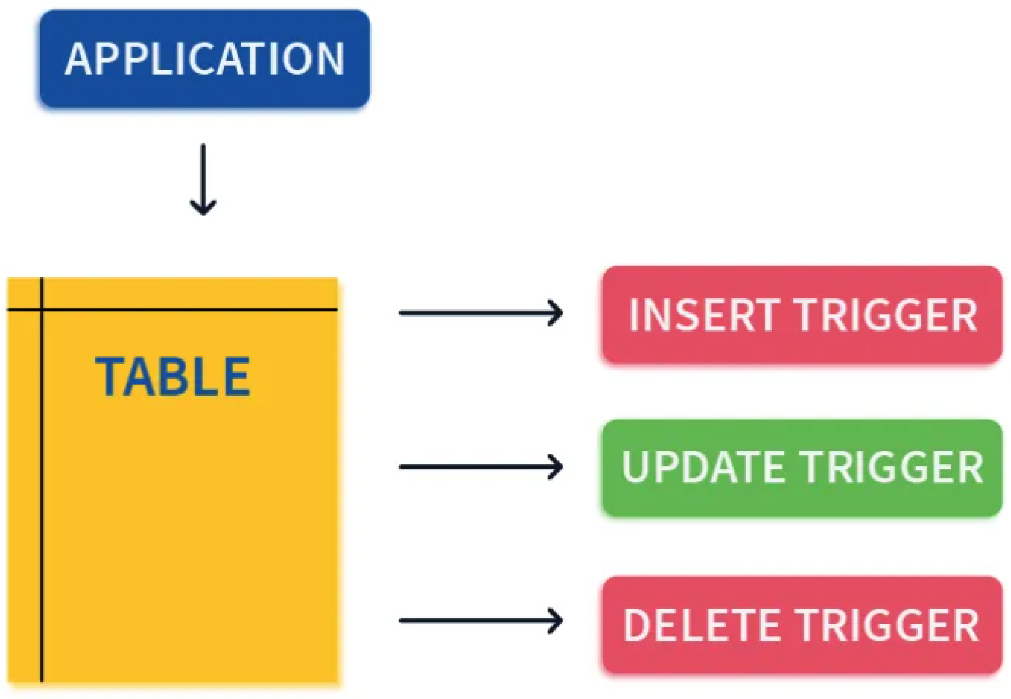
A SQL trigger is a special type of stored procedure. It is special because it is not called directly like a stored procedure. The main difference between a trigger and a stored procedure is that a trigger is called automatically when a data modification event is made against a table whereas a stored procedure must be called explicitly.

You might have encountered situations where you have to constantly check a database table for a particular column to get updated, where, once updated, you have some tasks to do.

Here the first solution which comes to mind is making a select statement (to check for a specific condition) on the table in an interval of a few seconds or minutes. But this doesn't seem to be a good choice, right? Instead of us asking the database whether the data has been updated or not, the database should tell us when the data gets updated. This is what exactly triggers do. Triggers, as the name tells us, are stored procedures (procedures are functions that contain SQL statements stored in the database and can return some output) that are executed or fired when some events occur. The user defines these events. The trigger may be set on a row insertion in a table or an update to an existing row and at other points, which we will discuss here. Before moving forward, it is important to discuss what DDL and DML operations are.

DDL stands for Data definition language, which is used in changing the structure of a table, i.e., creating a table, adding a column to an existing table, or deleting the whole table. However, DML stands for Data manipulation language, which is used to manipulate the data, i.e., insert some new data into the table or update existing data. Also, for deleting some rows, we make use of DML operations.

`Types of triggers:`



The following are the different types of triggers present in SQL.

DML Triggers

- These triggers fire in response to data manipulation language (DML) statements like INSERT, UPDATE, or DELETE.

After Triggers

- These triggers execute after the database has processed a specified event (such as an INSERT, UPDATE, or DELETE statement). AFTER triggers are commonly used to perform additional processing or auditing tasks after a data modification has occurred.

Instead Triggers

- These triggers are used for views and fire instead of the DML statement (INSERT, UPDATE, DELETE) on the view.

DDL Triggers

- These triggers fire in response to data definition language (DDL) statements like CREATE, ALTER, or DROP.

LOGON Triggers

- These triggers fire when a user logs into the database.

LOGOFF Triggers

- These triggers fire when a user logs out of the database.

SERVERERROR Triggers

- These triggers fire when a server error occurs.

Trigger Points of a SQL Trigger

1. When any DDL operation is done. E.g., CREATE, ALTER, DROP
2. For a DML operation. e.g., INSERT, UPDATE, DELETE.
3. For a database operation like LOGON, LOGOFF, STARTUP, SHUTDOWN or SERVERERROR

Syntax of Creating Triggers in SQL

We can create triggers for various types of operations, as discussed above. In this article, we will focus on DML triggers as these are the most commonly and extensively used triggers and also, at the same time, the most important. A DML trigger can be created by using the following syntax.

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
ON table_name
[FOR EACH ROW]
WHEN (condition)
[trigger_body]
```

Let’s discuss the different parts of the syntax:

- **CREATE [OR REPLACE] TRIGGER trigger_name:** In the first line, we give the name of the trigger we are creating/updating. Here [trigger_name] is to be replaced by the name you want to give to your trigger.
- **{BEFORE / AFTER / INSTEAD OF }:** Here, we define when the trigger would run, i.e., before or after the DML operation. For example, if we want a trigger to be executed after insertion to a table, we will write after here.
- **{INSERT [OR] / UPDATE [OR] / DELETE}:** This is the operation or event we have to define, which will trigger a procedure to be executed. If we want a procedure to run after a deletion happens on the table, then we will consider writing delete here.
- **on [table_name]:** Here, we have to specify the name of the table on which we are attaching the trigger. SQL will listen to changes on this table.
- **[for each row]:** This line specifies that the procedure will be executed for each one of the rows present. Here we can set a condition for which rows to be impacted. This part is optional, though; in case we don’t use this, the trigger shall convert to a “statement-level” trigger rather than being a “row-level” one, i.e., instead of firing the trigger procedure for each row, it will only execute once for each applicable statement.

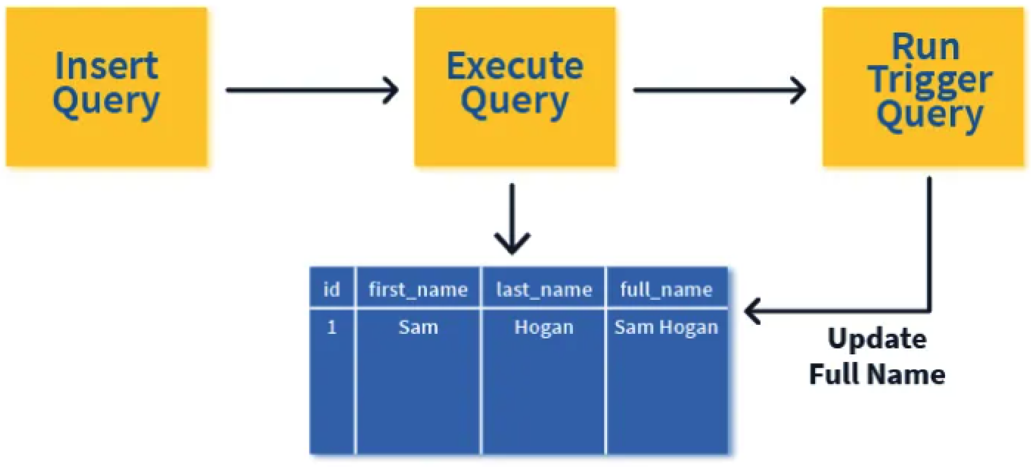
- **WHEN (condition):** Here, we mention some condition basis on which the trigger will run. In the absence of a when condition, you can expect it to run for all the eligible rows. This is very important as this will control which rows the trigger must run.
- **[trigger_body]:** This is the main logic of what to perform once the trigger is fired. In the previous statements, all we defined is when this trigger will be fired, but here we have to define what to do after the trigger is fired. This is the main execution code. Let’s take an example. Let’s assume a student table with column id, first_name, last_name, and full_name.

Let’s take an example. Let’s assume a student table with **column id**, **first_name, last_name, and full_name.**

`Query 1:`

```
CREATE TABLE student(Id integer PRIMARY KEY, first_name varchar(50), last_name
varchar(50), full_name varchar(50));
```

Here we will **create a trigger** to fill in the **full name** by **concatenating** the **first and last names**. So while inserting the values, we will only feed the first name and last name, and we will expect the **trigger** to **automatically update** each row with an additional column attribute bearing the full name.



First, let’s create a SQL Trigger –

`Query 2:`

```
create trigger student_name
after INSERT
on
student
for each row
BEGIN
```

```
   UPDATE student set full_name = first_name || ' ' || last_name;
END;
```

Here we can understand from the **trigger** query we have **set a trigger** after an insert is made to the table student. Once the **insert** is **done**, this **procedure** will be **fired**, which will run an **update** command to update the **student's** full names.

Let’s insert the students. `Query 3:`

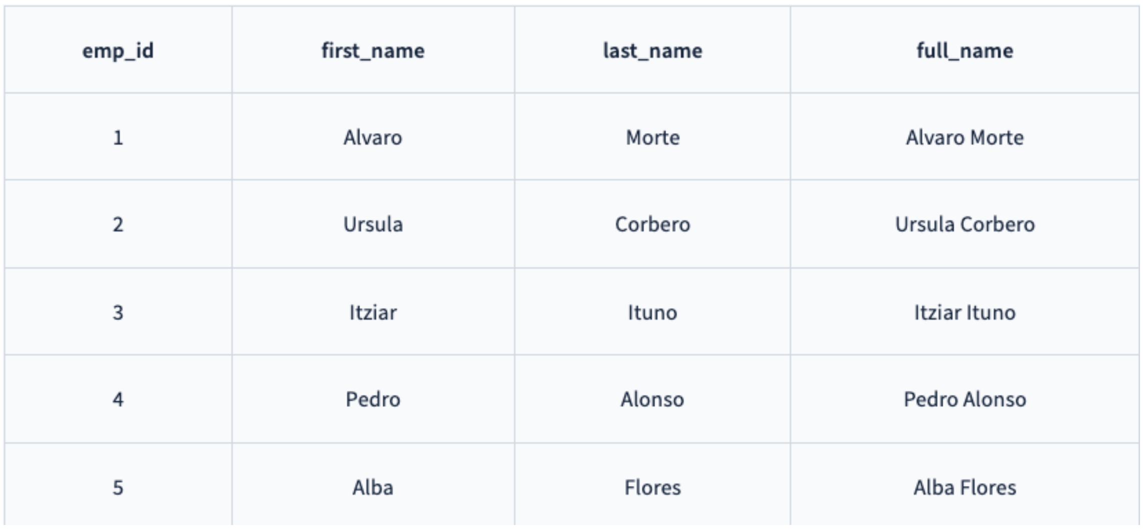
```
/* Create a few records in this table */
INSERT INTO student(id, first_name, last_name) VALUES(1,'Alvaro', 'Morte');
INSERT INTO student(id, first_name, last_name) VALUES(2,'Ursula', 'Corbero');
INSERT INTO student(id, first_name, last_name) VALUES(3,'Itziar', 'Ituno');
INSERT INTO student(id, first_name, last_name) VALUES(4,'Pedro', 'Alonso');
INSERT INTO student(id, first_name, last_name) VALUES(5,'Alba', 'Flores');
```

Here we have inserted five students’ data, and since we have a trigger created in our system to update the full_name, we are expecting the full name to be non-empty if we run a select query on this table.

`Query 4:`

```
/* Display all the records from the table */
SELECT * FROM student;
```

`Output:`



emp_id	first_name	last_name	full_name
1	Alvaro	Morte	Alvaro Morte
2	Ursula	Corbero	Ursula Corbero
3	Itziar	Ituno	Itziar Ituno
4	Pedro	Alonso	Pedro Alonso
5	Alba	Flores	Alba Flores

Here we can see since we had an update statement in the trigger procedure in query 2, the full names are automatically updated immediately after the inserts are done.

Please note that we have created an after trigger, which means the trigger will run after the insert is done (which is a DML operation).

Advantages of Triggers in SQL:

- Using triggers, you can eliminate the use of schedulers mostly. Schedulers are applications that keep running in the background and help run a task at a specific time. These timely checks can be removed and instead checked immediately whenever a particular data changes using triggers. It is basically a better replacement for schedulers.

- These are useful in having additional security checks. There are many critical components in business software, and some of them might risk highly sensitive data if not taken care of properly. Many developers work on a critical component, so some security checks must be done for the data being inserted, updated, or deleted. Here triggers help in maintaining a healthy system by warning about security issues if they exist.
- One can eliminate the risk of inserting invalid data into a column by having a check for that column or a simple update. Let’s say in the student_name table (where first_name and middle_name are both nullable, i.e., both the columns can accept null values), someone has inserted a row with middle_name keeping the first_name as null. Here the insertion will be successful, but the data is invalid. We can log the error, or also we can move the middle_name value to first_name.
- These are useful in cleaning activities since we can run some cleanup procedures on listening to deletion events. For example, there are two tables, students and parents. Ideally, if we deleted some students' data from the student's table, the corresponding data from parents (entries of those parents whose children are getting removed) also should get deleted. Since we are prone to such mistakes where we forget to delete data from parent's tables deleting from students, we can handle this very easily using triggers. We can have a trigger on students deletion, wherein on each DELETE operation, we may proceed to remove the corresponding entry from the parents' table.

Disadvantages of Triggers in SQL:

- These are difficult to troubleshoot because they are run automatically, and improper logging will result in unknown updates, which could be very time-consuming to debug in large-scale projects.
- Triggers increase the overhead of database DML queries. Since for every row we insert, update or delete, it may either do an update or an insertion – say into a new table, or even a row deletion altogether further that might not be required at the moment.
- Only limited validation can be done at triggers like `Not Null` checks, `Equality` checks, `Unique` checks, etc. Though we can use triggers to handle most of our business use cases, it doesn’t give a full proof solution for every case. Since SQL is a data manipulation language, we cannot expect it to replace programming language dependency for validations. We still need to depend on other validation along with this. Also, in some scenarios, data validation on SQL is costly compared to validation on the service side.