**TRIGGERS**

**Database Triggers**

======================

1. Database triggers are the database objects which reside in system catalog. The triggers are special type of procedures which can be called implicitly.

2. Each Trigger is associated with a table can be activated on any DML statement like (Insert, Update or Delete)

3. 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.

**Why Triggers**

=====================

1. Triggers help us to enforce business rules.

2. Triggers help us to validate data even before they are inserted or updated.

3. Triggers help us to keep log of records like maintaining audit trail.

4. SQL triggers provide an alternative way to check the integrity of data.

**Limitations of using Triggers**

=========================

1. SQL triggers are invoked and executed invisible from the client applications, therefore, it is difficult to figure out what happens in the database layer.

2. SQL triggers may increase the overhead of the database server.

**Row-Level Trigger:** It is a trigger, which is activated for each row by a triggering statement such as insert, update, or delete. For example, if a table has inserted, updated, or deleted multiple rows, the row trigger is fired automatically for each row affected by the [insert](https://www.javatpoint.com/mysql-insert), [update](https://www.javatpoint.com/mysql-update), or [delete statement](https://www.javatpoint.com/mysql-delete).

**Statement-Level Trigger:** It is a trigger, which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted.

**NOTE: We should know that MySQL doesn't support statement-level triggers. It provides supports for row-level triggers only.**

**Types of Triggers on the behalf of Activation Time:**

BEFORE INSERT – activated before data is inserted into the table.

AFTER INSERT – activated after data is inserted into the table.

BEFORE UPDATE – activated before data in the table is updated.

AFTER UPDATE – activated after data in the table is updated.

BEFORE DELETE – activated before data is removed from the table.

AFTER DELETE – activated after data is removed from the table.

|  |
| --- |
| ***When we use a statement that does not use INSERT, DELETE or UPDATE statement to change data in a table, the triggers associated with the table are not invoked.*** |

NAMING CONVENTION

We must use a unique name for each trigger associated with a table. However, We can have the same trigger name defined for different tables though it is a good practice.

**CREATE TRIGGER trigger\_name**

**{BEFORE | AFTER} {INSERT | UPDATE | DELETE}**

**ON table\_name**

**FOR EACH ROW**

**BEGIN**

**-- Trigger action: SQL statements to execute when the trigger fires**

**-- e.g., INSERT INTO another\_table, UPDATE some\_table, etc.**

**END;**

**TRIGGER\_TIME:** Specifies when the trigger is executed, either BEFORE or AFTER a row modification on the table. This is a required parameter.

**BEFORE :**

***Typically used to validate or modify data before it is committed to the table. For example, you might use a BEFORE trigger to check if a value meets certain criteria or to automatically modify data before it is inserted or updated.***

**AFTER :**

***AFTER triggers are suitable for actions that need to happen after the database operation has been committed, such as auditing, updating related records, or interacting with external systems.***

**TRIGGER\_EVENT:** Defines the operation that activates the trigger (INSERT, UPDATE, or DELETE). A trigger can only respond to one event; to handle multiple events, separate triggers are needed for each.

* The **NEW** and **OLD** modifiers are used to distinguish the colum values **BEFORE** and **AFTER** the execution of the DML statement.
* We can use the column name with NEW and OLD modifiers as **OLD.col\_name** and **NEW.col\_name**.
* **OLD.column\_name** indicates the column of an existing row before the updation or deletion occurs.
* **NEW.col\_name** indicates the column of a new row that will be inserted or an existing row after it is updated.

suppose we want to update the column name **message\_info** using the trigger. In the trigger body, we can access the column value before the update as **OLD.message\_info** and the new value **NEW.message\_info**.

**We can understand the availability of OLD and NEW modifiers with the below table:**

|  |  |  |
| --- | --- | --- |
| **Trigger Event** | **OLD** | **NEW** |
| INSERT | No | Yes |
| UPDATE | Yes | Yes |
| DELETE | Yes | No |

**Trigger Example :**

|  |
| --- |
| **CREATE TABLE employee(**  **name varchar(45) NOT NULL,**  **occupation varchar(35) NOT NULL,**  **working\_date date,**  **working\_hours varchar(10)**  **);** |

**INSERT** **INTO** employee **VALUES**

('Robin', 'Scientist', '2020-10-04', 12),

('Warner', 'Engineer', '2020-10-04', 10),

('Peter', 'Actor', '2020-10-04', 13),

('Marco', 'Doctor', '2020-10-04', 14),

('Brayden', 'Teacher', '2020-10-04', 12),

('Antonio', 'Business', '2020-10-04', 11);

Next, we will create a [**BEFORE INSERT trigger**](https://www.javatpoint.com/mysql-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**

use the following statements to invoke this trigger:

**INSERT** **INTO** employee **VALUES**

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

* we will get the output as on inserting the negative values into the working\_hours column of the table will automatically fill the zero value by a trigger.

**Show/List Triggers :**

**USE db\_name;**

**SHOW TRIGGERS;**

**OR**

**SHOW TABLES IN database\_name;**

**OR**

**SHOW TABLES FROM database\_name;**

**DROP TRIGGER :**

**DROP** **TRIGGER** [IF EXISTS] [schema\_name.]trigger\_name;

EXAMPLE :

**DROP** **TRIGGER** IF EXISTS employeedb.before\_update\_salaries;

**BEFORE INSERT TRIGGER :**

**SYNTAX:**

**CREATE TRIGGER trigger\_name BEFORE INSERT**

**ON table\_name FOR EACH ROW**

**BEGIN**

**variable declarations**

**trigger code**

**END**

**RESTRICTIONS :**

* We can access and change the **NEW** values only in a BEFORE INSERT trigger.
* We cannot access the **OLD** If we try to access the OLD values, we will get an error because OLD values do not exist.
* We cannot create a BEFORE INSERT trigger on a **VIEW**.

**AFTER INSERT TRIGGER :**

**SYNTAX:**

**CREATE TRIGGER trigger\_name AFTER INSERT**

**ON table\_name FOR EACH ROW**

**BEGIN**

**variable declarations**

**trigger code**

**END**

**RESTRICTIONS :**

* We can access the NEW values but cannot change them in an AFTER INSERT trigger.
* We cannot access the OLD If we try to access the OLD values, we will get an error because there is no OLD on the INSERT trigger.
* We cannot create the AFTER INSERT trigger on a VIEW.