1. MySQL Before Insert Trigger

In this article, we are going to learn how to create the first trigger in MySQL. We can create a new trigger in MySQL by using the CREATE TRIGGER statement. It is to ensure that we have trigger privileges while using the CREATE TRIGGER command. The following is the basic syntax to create a trigger:

1. **CREATE** **TRIGGER** trigger\_name  trigger\_time trigger\_event
2. **ON** table\_name **FOR** EACH ROW
3. **BEGIN**
4. --variable declarations
5. --trigger code
6. **END**;

### **Parameter Explanation**

The create trigger syntax contains the following parameters:

**trigger\_name:** It is the name of the trigger that we want to create. It must be written after the CREATE [TRIGGER statement](https://www.javatpoint.com/mysql-trigger). It is to make sure that the trigger name should be unique within the schema.

**trigger\_time:** It is the trigger action time, which should be either BEFORE or AFTER. It is the required parameter while defining a trigger. It indicates that the trigger will be invoked before or after each row modification occurs on the table.

**trigger\_event:** It is the type of operation name that activates the trigger. It can be either [INSERT](https://www.javatpoint.com/mysql-insert), [UPDATE](https://www.javatpoint.com/mysql-update), or [DELETE](https://www.javatpoint.com/mysql-delete) operation. The trigger can invoke only one event at one time. If we want to define a trigger which is invoked by multiple events, it is required to define multiple triggers, and one for each event.

**table\_name:** It is the name of the table to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.

**BEGIN END Block:** Finally, we will specify the statement for execution when the trigger is activated. If we want to execute multiple statements, we will use the BEGIN END block that contains a set of queries to define the logic for the trigger.

The trigger body can access the column's values, which are affected by the DML statement. The **NEW** and **OLD** modifiers are used to distinguish the column 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**. The 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.

### **MySQL Trigger Example**

Let us start creating a trigger in [MySQL](https://www.javatpoint.com/mysql-tutorial) that makes modifications in the employee table. First, we will create a new table named **employee** by executing the below statement:

**CREATE** **TABLE** employee(

**name** **varchar**(45) NOT NULL,

    occupation **varchar**(35) NOT NULL,

    working\_date **date**,

    working\_hours **varchar**(10)

);

Next, execute the below statement to **fill the records** into the employee table:

**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**.

DELIMITER //

**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** //

**DELIMITER ;**

Now, we can use the following statements to invoke this trigger:

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

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

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

('Alexander', 'Actor', '2020-10-012', -13);

# MySQL AFTER INSERT Trigger

After Insert Trigger in MySQL is invoked automatically whenever an insert event occurs on the table. In this article, we are going to learn how to create an after insert trigger with its syntax and example.

### **Syntax**

The following is the syntax to create an **AFTER INSERT** [trigger in MySQL](https://www.javatpoint.com/mysql-trigger):

1. **CREATE** **TRIGGER** trigger\_name
2. **AFTER** **INSERT**
3. **ON** table\_name **FOR** EACH ROW
4. trigger\_body ;

The AFTER INSERT trigger syntax parameter can be explained as below:

1. DELIMITER $$
2. **CREATE** **TRIGGER** trigger\_name **AFTER** **INSERT**
3. **ON** table\_name **FOR** EACH ROW
4. **BEGIN**
5. variable declarations
6. **trigger** code
7. **END**$$
8. DELIMITER ;

### **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**.

### **AFTER INSERT Trigger Example**

Let us understand how to create an AFTER INSERT trigger using the [**CREATE TRIGGER**](https://www.javatpoint.com/mysql-create-trigger) statement in [MySQL](https://www.javatpoint.com/mysql-tutorial) with an example.

Suppose we have created a table named "**student\_info**" as follows:

**CREATE** **TABLE** student\_info (

  stud\_id **int** NOT NULL,

  stud\_code **varchar**(15) **DEFAULT** NULL,

  stud\_name **varchar**(35) **DEFAULT** NULL,

  subject **varchar**(25) **DEFAULT** NULL,

  marks **int** **DEFAULT** NULL,

  phone **varchar**(15) **DEFAULT** NULL,

**PRIMARY** **KEY** (stud\_id)

) ;

Again, we will create a new table named **"student\_detail"** as follows:

**CREATE** **TABLE** student\_detail (

  stud\_id **int** NOT NULL,

  stud\_code **varchar**(15) **DEFAULT** NULL,

  stud\_name **varchar**(35) **DEFAULT** NULL,

  subject **varchar**(25) **DEFAULT** NULL,

  marks **int** **DEFAULT** NULL,

  phone **varchar**(15) **DEFAULT** NULL,

  Lasinserted **Time**,

**PRIMARY** **KEY** (stud\_id)

);

Next, we will use a CREATE TRIGGER statement to create an **after\_insert\_details** trigger on the **student\_info** table. This trigger will be fired after an insert operation is performed on the table.

DELIMITER //

**Create** **Trigger** after\_insert\_details

**AFTER** **INSERT** **ON** student\_info **FOR** EACH ROW

**BEGIN**

**INSERT** **INTO** student\_detail **VALUES** (new.stud\_id, new.stud\_code,

new.stud\_name, new.subject, new.marks, new.phone, CURTIME());

**END** //

**DELIMITER ;**

### **How to call the AFTER INSERT trigger?**

We can use the following statements to invoke the above-created trigger:

**INSERT** **INTO** student\_info **VALUES**

(10, 110, 'Alexandar', 'Biology', 67, '2347346438');

The table that has been modified after the update query executes is student\_detail. We can verify it by using the SELECT statement as follows:

**SELECT** \* **FROM** student\_detail;

# MySQL BEFORE DELETE Trigger

BEFORE DELETE Trigger in MySQL is invoked automatically whenever a delete operation is fired on the table. In this article, we are going to learn how to create a before delete trigger with its syntax and example.

### **Syntax**

The following is the syntax to create a BEFORE DELETE trigger in MySQL:

1. **CREATE** **TRIGGER** trigger\_name
2. BEFORE **DELETE**
3. **ON** table\_name **FOR** EACH ROW
4. Trigger\_body ;

DELIMITER $$

**CREATE** **TRIGGER** trigger\_name BEFORE **DELETE**

**ON** table\_name **FOR** EACH ROW

**BEGIN**

   variable declarations

**trigger** code

**END**$$

DELIMITER ;

### **Restrictions**

* We can access the OLD rows but cannot update them in a BEFORE DELETE trigger.
* We cannot access the NEW rows. It is because there are no new row exists.
* We cannot create a BEFORE DELETE trigger on a VIEW.

### **BEFORE DELETE Trigger Example**

Let us understand how to create a BEFORE DELETE trigger using the [CREATE TRIGGER statement in MySQL](https://www.javatpoint.com/mysql-create-trigger) with an example.

Suppose we have created a table named salaries to store the salary information of an employee as follows:

**CREATE** **TABLE** salaries (

    emp\_num **INT** **PRIMARY** **KEY**,

    valid\_from **DATE** NOT NULL,

    amount **DEC**(8 , 2 ) NOT NULL **DEFAULT** 0

);

Next, we will insert some records into this table using the below statement:

**INSERT** **INTO** salaries (emp\_num, valid\_from, amount)

**VALUES**

    (102, '2020-01-10', 45000),

    (103, '2020-01-10', 65000),

    (105, '2020-01-10', 55000),

    (107, '2020-01-10', 70000),

    (109, '2020-01-10', 40000);

Third, we will create another table named salary\_archives that keeps the information of deleted salary.

**CREATE** **TABLE** salary\_archives (

    id **INT** **PRIMARY** **KEY** AUTO\_INCREMENT,

    emp\_num **INT**,

    valid\_from **DATE** NOT NULL,

    amount **DEC**(18 , 2 ) NOT NULL **DEFAULT** 0,

    deleted\_time **TIMESTAMP** **DEFAULT** NOW()

);

We will then create a BEFORE DELETE trigger that inserts a new record into the salary\_archives table before a row is deleted from the salaries table.

DELIMITER $$

**CREATE** **TRIGGER** before\_delete\_salaries

BEFORE **DELETE**

**ON** salaries **FOR** EACH ROW

**BEGIN**

**INSERT** **INTO** salary\_archives (emp\_num, valid\_from, amount)

**VALUES**(OLD. emp\_num, OLD.valid\_from, OLD.amount);

**END**$$

DELIMITER ;

In this trigger, we have first specified the trigger name before\_delete\_salaries. Then, specify the triggering event. Third, we have specified the table name on which the trigger is associated. Finally, we have written the trigger logic inside the trigger body that insert the deleted row into the salary\_archives table.

**deleted\_time TIMESTAMP DEFAULT NOW()**:

* deleted\_time: This column is named deleted\_time and it stores timestamp values (TIMESTAMP).
* DEFAULT NOW(): This attribute sets the default value of the deleted\_time column to the current timestamp (NOW()) at the time a new row is inserted. If no value is provided for this column, the current timestamp will be used by default.

### **How to call the BEFORE DELETE trigger?**

Let us test the above created BEFORE DELETE trigger and how we can call them. So first, we will remove a row from the salaries table:

**DELETE** **FROM** salaries **WHERE** emp\_num = 105;

Second, we will query data from the salary\_archives table to verify the above-created trigger is invoked or not by using the select statement:

**SELECT** \* **FROM** salary\_archives;

After executing a statement, we can see that the trigger was invoked successfully and inserted a new record into the salary\_archives table.

Third, we will remove all rows from the salaries table:

**DELETE** **FROM** salaries;

Finally, we will query data from the salary\_archives table again. The trigger was called four times because the DELETE statement removed four records from the salaries table. See the below output:

**SELECT** \* **FROM** salary\_archives;

# MySQL AFTER DELETE Trigger

The AFTER DELETE Trigger in MySQL is invoked automatically whenever a delete event is fired on the table. In this article, we are going to learn how to create an AFTER DELETE trigger with its syntax and example.

### **Syntax**

The following is the syntax to create an **AFTER DELETE** trigger in MySQL:

1. **CREATE** **TRIGGER** trigger\_name
2. **AFTER** **DELETE**
3. **ON** table\_name **FOR** EACH ROW
4. Trigger\_body ;
5. DELIMITER $$
6. **CREATE** **TRIGGER** trigger\_name **AFTER** **DELETE**
7. **ON** table\_name **FOR** EACH ROW
8. **BEGIN**
9. variable declarations
10. **trigger** code
11. **END**$$
12. DELIMITER ;

### **Restrictions**

* We can access the OLD rows but cannot update them in the AFTER DELETE trigger.
* We cannot access the NEW rows. It is because there are no NEW row exists.
* We cannot create an AFTER DELETE trigger on a VIEW.

### **AFTER DELETE Trigger Example**

Let us understand how to create an AFTER DELETE trigger using the [CREATE TRIGGER statement in MySQL](https://www.javatpoint.com/mysql-create-trigger) with an example.

Suppose we have created a table named **salaries** to store the salary information of an employee as follows:

**CREATE** **TABLE** salaries (

    emp\_num **INT** **PRIMARY** **KEY**,

    valid\_from **DATE** NOT NULL,

    amount **DEC**(8 , 2 ) NOT NULL **DEFAULT** 0

);

Next, we will insert some records into this table using the below statement:

**INSERT** **INTO** salaries (emp\_num, valid\_from, amount)

**VALUES**

    (102, '2020-01-10', 45000),

    (103, '2020-01-10', 65000),

    (105, '2020-01-10', 55000),

    (107, '2020-01-10', 70000),

    (109, '2020-01-10', 40000);

Execute the SELECT query to see the table data.

SELECT \* from salaries;

Third, we will create another table named **total\_salary\_budget** that keeps the salary information from the salaries table.

**CREATE** **TABLE** total\_salary\_budget(

    total\_budget **DECIMAL**(10,2) NOT NULL

);

Fourth, we will use the **SUM()** function that returns the total salary from the salaries table and keep this information in the total\_salary\_budget table:

**INSERT** **INTO** total\_salary\_budget (total\_budget)

**SELECT** SUM(amount) **FROM** salaries;

SELECT \* from total\_salary\_budget;

We will then create an AFTER DELETE trigger that updates the total salary into the total\_salary\_budget table after a row is deleted from the salaries table.

DELIMITER $$

**CREATE** **TRIGGER** after\_delete\_salaries

**AFTER** **DELETE**

**ON** salaries **FOR** EACH ROW

**BEGIN**

**UPDATE** total\_salary\_budget **SET** total\_budget = total\_budget - old.amount;

**END**$$

DELIMITER ;

### **How to call the AFTER DELETE trigger?**

First, we will delete a salary from the salaries table using the following statements to invoke the above-created trigger:

**DELETE** **FROM** salaries **WHERE** emp\_num = 105;

Next, we will query data from the total\_salary\_budget table. We can see that table has been modified after the execution of the query. See the below output:

**SELECT** \* **FROM** total\_salary\_budget;

Third, we will remove all data from the salaries table:

**DELETE** **FROM** salaries;

SELECT \* from total\_salary\_budget;

# MySQL BEFORE UPDATE Trigger

BEFORE UPDATE Trigger in MySQL is invoked automatically whenever an update operation is fired on the table associated with the trigger. In this article, we are going to learn how to create a before update trigger with its syntax and example.

### **Syntax**

The following is the syntax to create a BEFORE UPDATE trigger in MySQL:

1. **CREATE** **TRIGGER** trigger\_name
2. BEFORE **UPDATE**
3. **ON** table\_name **FOR** EACH ROW
4. trigger\_body ;

### **Restrictions**

* We cannot update the OLD values in a BEFORE UPDATE trigger.
* We can change the NEW values.
* We cannot create a BEFORE UPDATE trigger on a VIEW.

### **BEFORE UPDATE Trigger Example**

Let us understand how to create a BEFORE UPDATE trigger using the [CREATE TRIGGER statement in MySQL](https://www.javatpoint.com/mysql-create-trigger) with an example.

Suppose we have created a table named **sales\_info** as follows:

**CREATE** **TABLE** sales\_info (

    id **INT** AUTO\_INCREMENT,

    product **VARCHAR**(100) NOT NULL,

    quantity **INT** NOT NULL **DEFAULT** 0,

    fiscalYear **SMALLINT** NOT NULL,

**CHECK**(fiscalYear BETWEEN 2000 and 2050),

**CHECK** (quantity >=0),

**UNIQUE**(product, fiscalYear),

**PRIMARY** **KEY**(id)

);

Next, we will insert some records into the sales\_info table as follows:

**INSERT** **INTO** sales\_info(product, quantity, fiscalYear)

**VALUES**

    ('2003 Maruti Suzuki',110, 2020),

    ('2015 Avenger', 120,2020),

    ('2018 Honda Shine', 150,2020),

    ('2014 Apache', 150,2020);

SELECT \* FROM sales\_info;

Next, we will use a **CREATE TRIGGER** statement to create a BEFORE UPDATE trigger. This trigger is invoked automatically before an update event occurs in the table.

DELIMITER $$

**CREATE** **TRIGGER** before\_update\_salesInfo

BEFORE **UPDATE**

**ON** sales\_info **FOR** EACH ROW

**BEGIN**

**DECLARE** error\_msg **VARCHAR**(255);

**SET** error\_msg = ('The new quantity cannot be greater than 2 times the current quantity');

    IF new.quantity > old.quantity \* 2 **THEN**

    SIGNAL SQLSTATE '45000'

**SET** MESSAGE\_TEXT = error\_msg;

**END** IF;

**END** $$

DELIMITER ;

### **// SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = error\_msg;**: It raises an exception with SQLSTATE '45000' (a generic error state) and sets the message text to the content of **error\_msg**.

**Common SQLSTATE Codes:**

* **'45000'**: General error for user-defined exceptions.
* **'01000'**: Warning messages (often used for informational purposes).
* **'23000'**: Integrity constraint violations (e.g., unique constraint violations).

DELIMITER $$

CREATE TRIGGER before\_insert\_salesInfo

BEFORE INSERT

ON sales\_info FOR EACH ROW

BEGIN

DECLARE error\_msg VARCHAR(255);

SET error\_msg = 'Fiscal year must be between 2000 and 2050';

IF NEW.fiscalYear < 2000 OR NEW.fiscalYear > 2050 THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = error\_msg;

END IF;

END $$

DELIMITER ;

**INSERT** **INTO** sales\_info(product, quantity, fiscalYear)

**VALUES**

    ('2003 Maruti Suzuki',110, 1919),

    ('2015 Avenger', 120,2020),

    ('2018 Honda Shine', 150,2020),

    ('2014 Apache', 150,2020);

SELECT \* FROM sales\_info;

### **How to call the BEFORE UPDATE trigger?**

First, we can use the following statements that update the quantity of the row whose id = 2:

**UPDATE** sales\_info **SET** quantity = 125 **WHERE** id = 2;

This statement works well because it does not violate the rule. Next, we will execute the below statements that update the quantity of the row as 600 whose id = 2

**UPDATE** sales\_info **SET** quantity = 600 **WHERE** id = 2;

# MySQL AFTER UPDATE TRIGGER

The AFTER UPDATE trigger in MySQL is invoked automatically whenever an UPDATE event is fired on the table associated with the triggers. In this article, we are going to learn how to create an AFTER UPDATE trigger with its syntax and example.

### **Syntax**

The following is the syntax to create an **AFTER UPDATE** trigger in MySQL:

1. **CREATE** **TRIGGER** trigger\_name
2. **AFTER** **UPDATE**
3. **ON** table\_name **FOR** EACH ROW
4. trigger\_body ;
5. DELIMITER $$
6. **CREATE** **TRIGGER** trigger\_name **AFTER** **UPDATE**
7. **ON** table\_name **FOR** EACH ROW
8. **BEGIN**
9. variable declarations
10. **trigger** code
11. **END**$$
12. DELIMITER ;

### **Restrictions**

* We can access the OLD rows but cannot update them.
* We can access the NEW rows but cannot update them.
* We cannot create an AFTER UPDATE trigger on a **VIEW**.

### **AFTER UPDATE Trigger Example**

Let us understand how to create an AFTER UPDATE trigger using the [CREATE TRIGGER statement in MySQL](https://www.javatpoint.com/mysql-create-trigger) with an example.

Suppose we have created a table named **students** to store the student's information as follows:

**CREATE** **TABLE** students(

    id **int** NOT NULL AUTO\_INCREMENT,

**name** **varchar**(45) NOT NULL,

    class **int** NOT NULL,

    email\_id **varchar**(65) NOT NULL,

**PRIMARY** **KEY** (id)

);

Next, we will insert some records into this table using the below statement:

**INSERT** **INTO** students (**name**, class, email\_id)

**VALUES** ('Stephen', 6, 'stephen@javatpoint.com'),

('Bob', 7, 'bob@javatpoint.com'),

('Steven', 8, 'steven@javatpoint.com'),

('Alexandar', 7, 'alexandar@javatpoint.com');

SELECT \* FROM STUDENTS;

**CREATE** **TABLE** students\_log(

    user **varchar**(45) NOT NULL,

    description **varchar**(65) NOT NULL

);

We will then create an AFTER UPDATE **trigger that promotes all students in the next class**, i.e., 6 will be 7, 7 will be 8, and so on. Whenever an updation is performed on a single row in the "**students**" table, a new row will be inserted in the **"students\_log**" table. This table keeps the **current user id** and a **description** regarding the current update. See the below trigger code.

DELIMITER $$

**CREATE** **TRIGGER** after\_update\_studentsInfo

**AFTER** **UPDATE**

**ON** students **FOR** EACH ROW

**BEGIN**

**INSERT** **into** students\_log **VALUES** (user(),

    CONCAT('Update Student Record ', OLD.**name**, ' Previous Class :',

    OLD.class, ' Present Class ', NEW.class));

**END** $$

DELIMITER ;

In this trigger, we have first specified the trigger name **after\_update\_studentsInfo**. Then, specify the triggering event. Third, we have specified the table name on which the trigger is associated. Finally, we have written the trigger logic inside the trigger body that performs updation in the "students" table and keeps the log information in the "students\_log" table.

### **How to call the AFTER UPDATE trigger?**

First, we will update the "students" table using the following statements that invoke the above-created trigger:

**UPDATE** students **SET** class = class + 1;

Next, we will query data from the **students** and **students\_log table**. We can see that table has been updated after the execution of the query. See the below output:

SELECT \* FROM STUDENTS;

Again, we will query data from the students\_log table that keeps the current user id and a description regarding the current update. See the below output:

SELECT \* FROM STUDENTS\_LOG;

A SQL view is a virtual table that is based on the result of a SQL query. It is stored in the database like a table, but it does not physically exist. Views are used to simplify complex queries, restrict access to sensitive data, and provide a consistent view of data to multiple users.

Creating a SQL view involves defining a SELECT statement and giving it a name. The SELECT statement can include joins, filters, and aggregations to retrieve the desired data. Once the view is created, it can be queried like a regular table, and any changes made to the underlying data will be reflected in the view.

Views can be used to combine multiple tables into a single logical table, hide complex queries from users, and provide a customized view of the data for different user roles. They can also be used to enforce data security by restricting access to certain columns or rows.

To create a view, the CREATE VIEW statement is used. Here is an example of creating a view that retrieves the names and ages of all employees:

CREATE VIEW EmployeeInfo AS

SELECT name, age

FROM employees;

Once the view is created, it can be queried like table:

SELECT \* FROM EmployeeInfo;

Views can also be modified or dropped using the ALTER VIEW and DROP VIEW statements, respectively.

Overall, SQL views are a powerful tool for simplifying queries, improving data security, and providing a consistent view of data to users.