MySQL Triggers: Examples, and Queries

Triggers are special database objects in MySQL that automatically execute specific SQL statements when certain events occur on a table.

Trigger Events:

- INSERT: Adding a new row to the table.
- UPDATE: Modifying existing data in a row.
- DELETE: Removing a row from the table.

Timing:

- BEFORE: Trigger executes before the event (can modify data or prevent the event itself).
- AFTER: Trigger executes after the event (useful for logging, cascading updates, etc.).

Functionality:

- Data validation: Ensure data adheres to specific rules (e.g., minimum price, valid email format).
- Auditing: Keep track of changes made to the table for record-keeping purposes.
- Cascading updates: Automatically update related tables when changes occur in one table.
- Custom actions: Perform any other tasks based on your application's logic.

Considerations:

- Performance: Triggers can add overhead, so use them judiciously when other approaches might be more efficient.
- Complexity: Triggers can become complex, so write them clearly and maintain them carefully.

The general syntax for creating a trigger is

```
DELIMITER //
CREATE TRIGGER <trigger_name>
{BEFORE | AFTER} #trigger_time
{INSERT | UPDATE | DELETE} #trigger_event
ON <table_name>
FOR EACH ROW
BEGIN
    <trigger_body>;
END;
//
```

Trigger Examples:

Here are some simple examples of triggers for each event and timing, along with their corresponding queries:

1. INSERT

a) BEFORE INSERT (Data Validation):

Consider the employee table having age column in that column, here's the query for the before insert trigger that sets the age to 0 if it's less than 0:

Example 01 :

```
DELIMITER //

CREATE TRIGGER age_check
BEFORE INSERT ON customers
FOR EACH ROW
BEGIN
  IF NEW.age < 0 THEN
    SET NEW.age = 0;
  END IF;
END;
//
```

Explanation:

- CREATE TRIGGER : This starts the trigger creation statement.
- `age_check`: This is the name you're giving to the trigger.
- BEFORE INSERT: This specifies that the trigger will execute before a new row is inserted into the customers table.
- FOR EACH ROW: This indicates that the trigger will fire for each individual row being inserted.
- IF `NEW.age < 0`, This is a conditional statement that checks if the age value is less than 0. If it is, the age is set to 0. Otherwise, the original age value is kept.
- SET `NEW.age = 0`: If the age is less than 0, the age is set to 0. Otherwise, the original age value is kept. This sets the value of the age column for the new row.

  Remember to replace `age_check` with your desired trigger name if you prefer something different.

Example 02:

```
DELIMITER //

CREATE TRIGGER price_check BEFORE INSERT ON products
FOR EACH ROW
BEGIN
  IF NEW.price <= 0 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Price
must be positive';
  END IF;
END;
//
```

Explanation:

- DELIMITER //: Sets a temporary delimiter to avoid conflicts with semicolons within the trigger body.
- CREATE TRIGGER: Creates a new trigger named price_check.
- BEFORE INSERT: Specifies the trigger fires before a new row is inserted into the products table.
- FOR EACH ROW: The trigger applies to each individual row being inserted.
- BEGIN: Start the trigger body.
- IF `NEW.price <= 0` THEN: Checks if the new price value is less than or equal to zero.
- SIGNAL `SQLSTATE '45000' SET MESSAGE_TEXT = 'Price must be positive'`: If the condition is true, raise an error with the specified code and message. This prevents the invalid insert.
- END IF: Ends the conditional block.
- END`;`: Ends the trigger body.
- DELIMITER `;`: Resets the default delimiter back to semicolon.

b) AFTER INSERT (Logging):

An after insert trigger in MySQL is a special program that automatically executes after a new row is inserted into a specific table. These triggers are useful for tasks like: Logging changes, Enforcing data integrity, Cascading updates, etc.

Example 01:

```
DELIMITER //

CREATE TRIGGER after_insert_audit AFTER INSERT ON customers
FOR EACH ROW
BEGIN
  INSERT INTO customer_audit_log (customer_id, action,
timestamp)
  VALUES (NEW.customer_id, 'Insert', NOW());
END; //
```

Explanation:

- CREATE TRIGGER : Starts creating a new trigger.
- `after_insert_audit`: The name given to the trigger.
- AFTER INSERT ON customers: Specifies that the trigger will fire after a new row is inserted into the customers table.
- FOR EACH ROW : Indicates that the trigger will execute for each individual row being inserted.
- BEGIN: Marks the beginning of the trigger body.
- INSERT INTO...VALUES...: : Inserts a new row into the customer_audit_log table, recording the customer ID, action (`'Insert'`), and timestamp of the insertion.
- END; : Terminates the trigger body.

2. UPDATE:

a) BEFORE UPDATE :

An update before trigger in MySQL executes before any changes are applied to a row that is being updated in a specific table. These triggers are beneficial for tasks like: Enforcing data integrity, Auditing changes,Cascading updates.

Here's the general syntax for creating an update before trigger:

```
CREATE TRIGGER <trigger_name>
BEFORE UPDATE ON <table_name>
FOR EACH ROW
<trigger_body>
```

Example 01 :

```
DELIMITER //
CREATE TRIGGER before_update_cust
BEFORE UPDATE ON customers
FOR EACH ROW
BEGIN
  IF NEW.email <> OLD.email THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Email
cannot be changed';
  END IF;
END; //
```

Explanation:

- DELIMITER //: Temporarily changes the statement delimiter to "//" to accommodate semicolons within the trigger body.
- CREATE TRIGGER: Creates the trigger named before_update_cust.
- BEFORE UPDATE ON `customers`: Specifies that the trigger will fire before any update on the customers table.

- FOR EACH ROW: Indicates the trigger will execute for each updated row.
- BEGIN: Marks the start of the trigger's actions.
- `IF NEW.email <> OLD.email THEN`: Checks if the new email (NEW.email) is different from the original email (OLD.email).
- `SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Email cannot be changed';` : If the email has been changed, it raises an error with the specified code and message, preventing the update.
- END IF: Ends the conditional block.
- END;: Terminates the trigger body.
- //: Ends the temporary delimiter change.

Now if you try to update the table, it will trigger the above query and give you the msg as mentioned in the trigger.

Try this one :

```
UPDATE customers
SET email = 'john.doe02@example.com'
WHERE customer_id = 1;
```

b) AFTER UPDATE :

An update after trigger in MySQL is a special program that automatically executes after a row is updated in a specific table. These triggers are useful for various tasks, such as: Auditing changes, Cascading updates, Custom actions

Here's the general syntax for creating an update after trigger:

```
CREATE TRIGGER <trigger_name>
AFTER UPDATE ON <table_name>
FOR EACH ROW
<trigger_body>
```

Example 01:

```
DELIMITER //
CREATE TRIGGER after_update_cust
AFTER UPDATE ON customers
FOR EACH ROW
BEGIN
  INSERT INTO customer_audit_log (customer_id, action,
timestamp)
  VALUES (NEW.customer_id, 'Update', NOW());
END; //
DELIMITER ;
```

Explanation:

- DELIMITER //: This line temporarily sets the statement delimiter to "//" to avoid conflicts with semicolons within the trigger body.
- CREATE TRIGGER `after_update_cust`: This line starts creating a new trigger named after_update_cust.
- AFTER UPDATE ON `customers`: This line specifies that the trigger will fire after any update to the customers table.
- FOR EACH ROW: This line indicates that the trigger will execute for each individual row being updated.
- BEGIN : This line marks the beginning of the trigger body.
- INSERT INTO...VALUES... : This line inserts a new row into the `customer_audit_log` table. The inserted data includes:

    - `customer_id`: The ID of the updated customer, retrieved using `NEW.customer_id`.
    - `action`: A string indicating the action that happened, which is `"Update"` in this case.
    - `timestamp`: The current date and time when the trigger executed, retrieved using NOW().
- END;: This line terminates the trigger body.
- //: This line ends the temporary delimiter change.

This trigger will now log every update made to the customers table, keeping track of who made the changes, when they were made, and which customer was affected.

3. DELETE:

a) BEFORE DELETE :

Before delete trigger is a database object that gets executed automatically before a row is deleted from a table. Triggers are typically used to enforce business rules, perform validation, or carry out additional actions when certain events occur on a table.

If you want to record information about deleted customers before they are removed from the database, you can create a trigger that inserts data into a separate audit table:

Consider that you are having customer_log table consist of columns `customer_id, name, email, phone, deleted_at`

```
DELIMITER //
CREATE TRIGGER customer_before_delete
BEFORE DELETE ON customers
FOR EACH ROW
BEGIN
  INSERT INTO customer_log (customer_id, name, email, phone,
deleted_at)
  VALUES (OLD.customer_id, OLD.name, OLD.email, OLD.phone,
NOW());
END;
// DELIMITER ;
```

This trigger creates a record in the customer_audit_log table whenever a row is deleted from the customers table, capturing the customer's ID, name, email, phone number, and the deletion timestamp.

Explanation:

- BEGIN: This marks the start of the trigger body, where the actual actions will be executed.
- INSERT: This statement inserts a new record into the `customer_log` table.
- VALUES: This specifies the values for each column in the new record. `OLD` refers to the original row being deleted from the customers table.
- NOW(): This function retrieves the current timestamp and inserts it into the `deleted_at` column.
- END;: This marks the end of the trigger body.

b) AFTER DELETE :

In database systems, an after delete trigger is a special type of database trigger that automatically executes a set of predefined actions after one or more rows are deleted from a specified table. This trigger serves the purpose of auditing or maintaining data integrity when deletions occur

This trigger can perform tasks after a customer is deleted

```
DELIMITER //

CREATE TRIGGER after_delete_cust
AFTER DELETE ON customers
FOR EACH ROW
BEGIN
  INSERT INTO customer_audit_log (customer_id, action,
timestamp)
  VALUES (OLD.customer_id, 'Delete', NOW());
END; //

DELIMITER ;
```

Explanation:

- When a DELETE statement removes a row from the customers table:
- The `after_delete_cust` trigger fires for each deleted row.
- For each row, an INSERT statement is executed.
- The `customer_id` of the deleted customer, retrieved from the OLD row, is inserted into the `customer_audit_log` table.
- The action is recorded as `"Delete"`.
- The current timestamp is captured and stored.