13 MySQL Triggers

A trigger in SQL is a special type of stored procedure that is automatically executed or fired when certain events occur within a database table or view. Triggers are used to enforce business rules, maintain data integrity, and automate certain actions in response to changes in the database.

Syntax for Creating a Trigger

```SQL
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
   -- Trigger logic here
END;
```

Key Components

- `trigger_name`: The name of the trigger.
- `BEFORE` / `AFTER`: Specifies when the trigger is activated. `BEFORE` means the trigger fires before the operation is executed, and `AFTER` means it fires after the operation.
- `INSERT | UPDATE | DELETE`: The type of operation that activates the trigger.
- `table_name`: The table to which the trigger is attached.
- `FOR EACH ROW`: Indicates that the trigger will execute once for each row affected by the operation.

Triggers can be powerful tools for maintaining data integrity and automating tasks, but they should be used carefully to avoid performance issues and unintended consequences.

NOTE:

> You can't manually call or invoke a trigger. It is automatically invoked before or after the `INSERT`, `UPDATE`, or `DELETE` operation on a table.

Use cases | Examples

01. Inserting old data to a table

SQL

```sql
create table sales(id int auto_increment primary key,
                   product varchar(30) not null,
                   Price numeric(10,2));


create table sales_update_track(
   id int primary key auto_increment,
   product_id int not null,
    changed_at timestamp,
   before_price numeric(10,2) not null,
    after_price numeric(10,2) not null);


insert into sales (id, product, price)
values
(1, "Coffe", 20),
(2, "Maggie", 60),
(3, "Juice", 70);

select * from sales;

delimiter $$

create trigger before_sales_update before update on sales for each row
begin

   insert into sales_update_track(product_id, changed_at, before_price,
after_price)
       value (old.id, now(), old.price, new.price);

end$$


delimiter ;

update sales set price = 100 where id = 3;


select * from sales_update_track;
```

02. Employee Log

```sql
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255)
);

CREATE TABLE log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    employee_id INT,
    old_name VARCHAR(255),
    new_name VARCHAR(255),
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create a Trigger

Create a trigger that logs changes to the `employees` table.

```sql
DELIMITER $$

CREATE TRIGGER log_employee_update
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO log (employee_id, old_name, new_name)
    VALUES (OLD.id, OLD.name, NEW.name);
END$$

DELIMITER ;
```

Step 3: Update the Table to Trigger the Trigger

When you update the `employees` table, the trigger is automatically called.

```sql
-- Insert an initial record
INSERT INTO employees (name) VALUES ('John Doe');

-- Update the record
UPDATE employees SET name = 'Jane Doe' WHERE id = 1;
```

Step 4: Check the Log Table

The `log` table should now have an entry for the update operation.

```sql
SELECT * FROM log;
```

Summary

- **Enforcing Business Rules**: Automatically set timestamps, enforce constraints.
- **Auditing and Logging**: Maintain detailed logs of changes.
- **Maintaining Referential Integrity**: Implement cascading deletes/updates.
- **Synchronizing Tables**: Keep related tables in sync.
- **Complex Validation**: Enforce advanced business rules and constraints.
- **Derived or Calculated Fields**: Automatically calculate and store derived values.
- **Notification Systems**: Implement notification mechanisms by writing to a queue or log.

By utilizing triggers in these ways, you can ensure data integrity, automate repetitive tasks, and enforce business logic directly at the database level, thus maintaining a cleaner and more reliable data environment.