SQL Triggers: A Detailed Explanation

What is a Trigger in SQL?

A **trigger** in SQL is a special type of stored procedure that automatically executes or fires when a specific event occurs in the database. This event could be an `INSERT`, `UPDATE`, or `DELETE` operation on a table. Triggers are used to enforce business rules, validate data integrity, or log changes automatically without user intervention.

Types of SQL Triggers

1. **Before Trigger**: Executes **before** the event (e.g., `BEFORE INSERT`).
2. **After Trigger**: Executes **after** the event (e.g., `AFTER INSERT`).
3. **Instead of Trigger** (some databases): Replaces the action with a custom behavior.

Syntax for Creating a Trigger

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON table_name
FOR EACH ROW
BEGIN
   -- SQL statements to be executed
END;
```

Why Use Triggers?

Triggers are used for:

1. **Automatically Logging Changes**: Keep a log of changes in the data (audit trail).
2. **Enforcing Business Rules**: Ensure that certain conditions are met before modifying data.
3. **Maintaining Referential Integrity**: Ensure data consistency across tables.
4. **Cascading Operations**: Perform automatic updates or deletions in related tables.

Real-Life Example: Employee Salary History

Consider a scenario where you have an `employee` table. You want to keep a history of any salary changes made to employees. Instead of manually logging changes every time the salary is updated, you can use a trigger to do this automatically.

Employee Table (`employee`)

employee_id	name	age	salary
1	John	28	50000
2	Sarah	32	65000
3	Michael	45	75000

Salary History Table (`salary_history`)

history_id	employee_id	old_salary	new_salary	change_date
1	1	50000	55000	2024-09-20 10:30:00

The `salary_history` table is designed to track every change in an employee's salary. We want to create a trigger that automatically inserts a record into the `salary_history` table whenever the salary in the `employee` table is updated.

Creating the Trigger

Here is how you would create a trigger to log salary changes:

```
CREATE TRIGGER log_salary_update
AFTER UPDATE ON employee
FOR EACH ROW
BEGIN
   -- Insert the old and new salary into salary_history
   INSERT INTO salary_history (employee_id, old_salary, new_salary,
change_date)
   VALUES (OLD.employee_id, OLD.salary, NEW.salary, NOW());
END;
```

Explanation of Trigger Components

1. **Trigger Name**: `log_salary_update` is the name of the trigger.
2. **AFTER UPDATE**: The trigger fires **after** an update on the `employee` table.
3. **FOR EACH ROW**: The trigger will execute for **each row** that is updated.
4. **OLD.salary**: Refers to the value of `salary` before the update.
5. **NEW.salary**: Refers to the updated value of `salary`.
6. **NOW()**: Captures the current timestamp when the update happens.

How the Trigger Works

Whenever the `salary` column in the `employee` table is updated, the trigger automatically logs the old and new salary values into the `salary_history` table, along with the `employee_id` and the time of the change.

Example Update

Suppose John's salary is updated from 50,000 to 55,000. The following query is executed:

```
UPDATE employee
SET salary = 55000
WHERE employee_id = 1;
```

After this update, the trigger `log_salary_update` automatically inserts a new record into the `salary_history` table:

history_id	employee_id	old_salary	new_salary	change_date
2	1	50000	55000	2024-10-24 11:00:00

Advantages of Triggers

1. **Automatic Execution**: Triggers execute automatically in response to an event, which ensures that business rules are always enforced.
2. **Audit Trails**: Triggers can automatically log changes in a table, helping to maintain an audit trail without manual intervention.
3. **Enforcing Complex Constraints**: Triggers can enforce more complex business rules that cannot be implemented using just constraints (like `CHECK`, `NOT NULL`).
4. **Data Consistency**: Helps maintain data consistency by performing automatic updates across related tables (e.g., cascading deletes).

Disadvantages of Triggers

1. **Performance Overhead**: Triggers can slow down the performance of the database, especially when performing operations on large datasets, as they execute every time the event occurs.
2. **Complexity**: Triggers can sometimes make debugging harder because they can alter data or behavior behind the scenes without clear visibility.
3. **Limited to Certain Operations**: Triggers can only be set on specific events (`INSERT`, `UPDATE`, `DELETE`), and not on operations like `SELECT`.
4. **Potential for Infinite Loops**: If not designed carefully, triggers can create a loop of events (e.g., an `UPDATE` trigger that updates the table, firing the trigger again).

Disabling and Dropping Triggers
If you want to temporarily stop a trigger from executing, you can disable it:

```
ALTER TABLE employee DISABLE TRIGGER log_salary_update;
```

To remove a trigger permanently:

```
DROP TRIGGER log_salary_update;
```

Modifying a Trigger

To modify an existing trigger, you must drop and recreate it with the new logic.

Other Practical Applications of Triggers

1.Maintaining a Log Table: Automatically track any `INSERT`, `UPDATE`, or `DELETE` on important tables by creating log tables.

Example: Create a trigger to log every `DELETE` on a `customer` table.

```
CREATE TRIGGER log_customer_delete
AFTER DELETE ON customer
FOR EACH ROW
BEGIN
   INSERT INTO customer_deletions (customer_id, deleted_at)
   VALUES (OLD.customer_id, NOW());
END;
```

1. **Enforcing Referential Integrity**: When a parent record is deleted, a trigger can ensure that all related child records are either deleted or updated.
2. **Automatic Calculations**: If an `order` is updated with new item quantities, a trigger can automatically update the total order price in an `order_summary` table.

Key Points About Triggers

1. **Definition**: A trigger is a special kind of stored procedure that automatically executes when specific database events occur (like `INSERT`, `UPDATE`, `DELETE`).
2. **Benefits**: Automates logging, enforces business rules, maintains data integrity, and triggers cascading operations.
3. **Creation**: Use `CREATE TRIGGER` to define triggers. Triggers can run before or after the event (`BEFORE INSERT`, `AFTER UPDATE`, etc.).
4. **Example**: You can create a trigger to automatically log salary changes in an `employee` table whenever a salary is updated.

5. **Caution**: Triggers can affect performance, add complexity, and in some cases, introduce unintended behaviours like infinite loops.
6. **Disabling and Dropping**: Triggers can be temporarily disabled or permanently dropped when needed.