**What is a Trigger in Oracle SQL?**

A **trigger** in Oracle SQL is a special type of stored procedure that automatically **fires** (executes) when a specific **event** occurs on a table or view. The event can be an **INSERT**, **UPDATE**, or **DELETE** operation.

Triggers are useful for enforcing business rules, auditing, or automatically performing certain actions when data changes in the database.

**Uses of Triggers:**

1. **Enforcing Data Integrity**: You can use triggers to enforce business rules or constraints beyond what is possible with standard constraints.
2. **Auditing**: Triggers can be used to track changes in data by logging insertions, updates, or deletions.
3. **Enforcing Referential Integrity**: Triggers can help maintain integrity between tables, for example, when updates or deletes happen.
4. **Automatic Computation**: You can use triggers to automatically calculate values or perform additional tasks (e.g., updating stock quantity, auditing changes).

**Types of Triggers in Oracle:**

1. **DML Triggers** (Data Manipulation Language Triggers):
   * **BEFORE** Trigger: Fires before the actual SQL statement (INSERT, UPDATE, DELETE) is executed.
   * **AFTER** Trigger: Fires after the actual SQL statement (INSERT, UPDATE, DELETE) is executed.
   * **INSTEAD OF** Trigger: Used for views; it fires instead of the DML operation.
2. **DDL Triggers** (Data Definition Language Triggers): Used to monitor changes to the structure of database objects (like tables, indexes, etc.).
3. **LOGON and LOGOFF Triggers**: Fires when a user logs into or out of the database.
4. **Compound Triggers**: These are a combination of multiple triggers that fire in a specific order to reduce dependency issues between triggers.

**Syntax for Creating a Trigger**

The basic syntax for creating a trigger in Oracle SQL is:

CREATE [OR REPLACE] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF}

{INSERT | UPDATE | DELETE}

ON table\_name

[FOR EACH ROW] -- Only for row-level triggers

DECLARE

-- Variable declarations (optional)

BEGIN

-- Trigger logic (SQL statements)

END;

* **trigger\_name**: Name of the trigger.
* **BEFORE / AFTER**: Specifies when the trigger should fire (before or after the DML operation).
* **INSERT / UPDATE / DELETE**: Specifies the DML operation that will activate the trigger.
* **FOR EACH ROW**: Specifies that the trigger will be executed for each row affected by the operation (this is for **row-level** triggers).
* **table\_name**: The table or view that the trigger is associated with.

**Example 1: Creating a BEFORE INSERT Trigger**

Let's create a trigger that ensures the salary field in the employees table is never set to a value less than 1000.

**Step 1: Creating the Trigger**

CREATE OR REPLACE TRIGGER check\_salary\_before\_insert

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

IF :NEW.salary < 1000 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Salary cannot be less than 1000');

END IF;

END;

**Explanation:**

* **BEFORE INSERT**: The trigger fires before a new row is inserted into the employees table.
* **FOR EACH ROW**: The trigger executes for each row that is being inserted.
* **:NEW.salary**: Refers to the new value of the salary column being inserted. :NEW is used in row-level triggers to access the value of the row being inserted, updated, or deleted.
* **RAISE\_APPLICATION\_ERROR**: This is used to raise a custom error if the condition is violated (if salary is less than 1000).

**Step 2: Testing the Trigger**

Try to insert a row with a salary less than 1000:

INSERT INTO employees (employee\_id, first\_name, last\_name, salary)

VALUES (1001, 'John', 'Doe', 900);

**Expected Output:**

ORA-20001: Salary cannot be less than 1000

The trigger prevents the insertion of this row because the salary is less than the specified limit.

**Example 2: Creating an AFTER-UPDATE Trigger**

Suppose you want to track any changes made to the salary field in the employees table and log these changes into an salary\_audit table.

**Step 1: Create an salary\_audit Table**

CREATE TABLE salary\_audit (

audit\_id NUMBER PRIMARY KEY,

employee\_id NUMBER,

old\_salary NUMBER,

new\_salary NUMBER,

change\_date DATE

);

**Step 2: Creating the Trigger**

CREATE OR REPLACE TRIGGER track\_salary\_change

AFTER UPDATE ON employees

FOR EACH ROW

BEGIN

IF :OLD.salary != :NEW.salary THEN

INSERT INTO salary\_audit (audit\_id, employee\_id, old\_salary, new\_salary, change\_date)

VALUES (salary\_audit\_seq.NEXTVAL, :NEW.employee\_id, :OLD.salary, :NEW.salary, SYSDATE);

END IF;

END;

**Explanation:**

* **AFTER UPDATE**: The trigger fires after an update operation on the employee’s table.
* **:OLD.salary**: Refers to the old salary value before the update.
* **:NEW.salary**: Refers to the new salary value after the update.
* **INSERT INTO salary\_audit**: This inserts a record into the salary\_audit table every time the salary is changed.
* **salary\_audit\_seq.NEXTVAL**: This generates a new unique ID for the audit entry using a sequence (you should have created a sequence salary\_audit\_seq).

**Step 3: Testing the Trigger**

Let's update an employee's salary:

UPDATE employees

SET salary = 12000

WHERE employee\_id = 1001;

**Step 4: Check the salary\_audit Table**

SELECT \* FROM salary\_audit;

**Expected Output:**

AUDIT\_ID EMPLOYEE\_ID OLD\_SALARY NEW\_SALARY CHANGE\_DATE

------------------------------------------------------------------------------------------------

1 1001 900 12000 2024-12-10

The trigger successfully captured the change in salary and inserted the change into the salary\_audit table.

**Example 3: INSTEAD OF Trigger for a View**

Let's say you have a view called employee\_view that combines data from the employees and departments tables, and you want to allow updates to the view (which normally isn't allowed).

**Step 1: Create a View**

CREATE VIEW employee\_view AS

SELECT e.employee\_id, e.first\_name, e.salary, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id;

**Step 2: Create an INSTEAD OF Trigger**

CREATE OR REPLACE TRIGGER update\_employee\_view

INSTEAD OF UPDATE ON employee\_view

FOR EACH ROW

BEGIN

UPDATE employees

SET salary = :NEW.salary

WHERE employee\_id = :OLD.employee\_id;

END;

**Explanation:**

* **INSTEAD OF**: This trigger fires instead of the normal UPDATE operation on the employee\_view.
* **FOR EACH ROW**: The trigger is executed for each row being updated in the view.
* **UPDATE employees**: Instead of updating the view, it directly updates the underlying employees table.

**Step 3: Test the Trigger**

UPDATE employee\_view

SET salary = 11000

WHERE employee\_id = 1001;

**Step 4: Check the Employees Table**

SELECT \* FROM employees WHERE employee\_id = 1001;

**Expected Output:**

EMPLOYEE\_ID FIRST\_NAME SALARY DEPARTMENT\_ID

----------------------------------------------

1001 John 11000 10

The trigger performs the update on the employees table instead of the view.

**Conclusion**

Triggers in Oracle SQL are powerful tools for automating actions in the database when data changes occur. They can:

* Enforce business rules.
* Audit changes.
* Automate calculations or updates.
* Work with views and tables.

Triggers help ensure data integrity, enforce consistency, and can reduce the need for manual intervention or additional application logic.

Top of Form

Bottom of Form