Triggers

1. Structure and Components

A trigger definition fundamentally consists of three parts:

Component	Description
Event	The specific Data Manipulation Language (DML) or Data Definition Language (DDL) action that causes the trigger to fire.
Timing	Specifies whether the trigger executes **BEFORE** or **AFTER** the initiating event.
Action	The set of SQL statements and logic that the database executes when the trigger fires. This is the body of the stored procedure.

2. Event and Timing (The When)

The combination of event and timing dictates precisely when a trigger's code runs.

Events

- **DML:** `INSERT`, `UPDATE`, `DELETE`.
- **DDL:** `CREATE`, `ALTER`, `DROP` (used for database-level auditing).
- **Database Operations:** `LOGON`, `LOGOFF`, `STARTUP`, `SHUTDOWN`.

Timing

- **BEFORE Trigger:** Fires *before* the changes are applied to the database.
  - **Use Case:** Ideal for **data validation** (checking if data meets certain criteria) or **data modification** (setting a default value or updating a value) before the row is permanently written.
- **AFTER Trigger:** Fires *after* the changes are successfully applied to the database.
  - **Use Case:** Ideal for **logging/auditing** and **cascading actions** (propagating changes to other tables, like updating an inventory count).

3. Granularity (The Scope)

Triggers are also categorized by their granularity:

- **ROW-LEVEL Trigger (`FOR EACH ROW`):** The trigger body executes once for **every row** affected by the event.
    - *Example:* An `UPDATE` statement modifies 50 rows; the row-level trigger fires 50 times. Best for data validation and fine-grained auditing.
- **STATEMENT-LEVEL Trigger (`FOR EACH STATEMENT`):** The trigger body executes only once for the **entire SQL statement**, regardless of how many rows are affected (even if zero).
    - *Example:* An `UPDATE` statement modifies 50 rows; the statement-level trigger fires only once. Best for logging the operation itself or performing security checks.

4. Special Row Variables (The Data)

Within a trigger's code, two special "virtual tables" or row variables are typically available to inspect the data involved in the operation:

Variable	DML Event	Purpose
:NEW	`INSERT` , `UPDATE`	Holds the value of the **new row** or the data **after** the update.
:OLD	`UPDATE` , `DELETE`	Holds the value of the **old row** or the data **before** the deletion/update.

Suitable Examples

1. Example: Enforcing a Complex Integrity Constraint (BEFORE Trigger)

Triggers are essential for constraints that span multiple records or tables.

Scenario: In a bank's `Accounts` table, we want to prevent a transaction that would make the account balance negative.

Component	Definition
Event	`UPDATE` (when the `balance` is updated).
Timing	`BEFORE` (check balance *before* the update is committed).
Granularity	`ROW-LEVEL` (check each account individually).
Action	If the new balance (`:NEW.balance`) is less than zero, cancel the operation and raise an error.

Conceptual Code (using Oracle/PostgreSQL syntax):

SQL

```
CREATE OR REPLACE TRIGGER Check_Negative_Balance
BEFORE UPDATE OF balance ON Accounts
FOR EACH ROW
WHEN (NEW.balance < 0)
BEGIN
  -- This action will fire only if the NEW balance is negative.
  RAISE_APPLICATION_ERROR(-20001, 'Transaction failed: Account balance cannot be negative.');
END;
```

2. Example: Auditing and Logging Changes (AFTER Trigger)

Triggers are the primary mechanism for database auditing.

Scenario: Record every time an employee's salary is updated in the `Employees` table into a separate `AuditLog` table.

Component	Definition
Event	`UPDATE` (when any column in `Employees` is updated).
Timing	`AFTER` (log the change after it is successfully made).
Granularity	`ROW-LEVEL` (log the change for each affected employee).
Action	Insert the old salary, new salary, and timestamp into the `AuditLog` table.

Conceptual Code (using standard syntax principles):

SQL

```
CREATE TRIGGER Audit_Salary_Changes
AFTER UPDATE ON Employees
FOR EACH ROW
WHEN (OLD.salary <> NEW.salary)  -- Condition to fire only if salary actually changed
BEGIN
  INSERT INTO AuditLog (employee_id, old_salary, new_salary, change_date)
  VALUES (:OLD.employee_id, :OLD.salary, :NEW.salary, CURRENT_TIMESTAMP);
END;
```