TRIGGERS

A trigger is a PL/SQL block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed. A trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs. It is a mechanism to ease out the task of implementing organization's business rules.

Like a stored procedure, a trigger is a named PL/SQL block that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, user can enable and disable a trigger, but cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes it — that is, the trigger fires — whenever its triggering event occurs. While a trigger is disabled, it does not fire.

While creating a trigger with the CREATE TRIGGER statement, the user should -

- Specify the triggering event in terms of triggering statements and the item on which they act. The trigger is said to be created on or defined on the item, which is either a table, a view, a schema, or the database.
- Specify the timing point, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects.

By default, a trigger is created in the enabled state.

If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a **DML trigger**.

If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a **System Trigger**.

A **conditional trigger** has a WHEN clause that specifies a SQL condition that the database evaluates for each row that the triggering statement affects.

When a trigger fires, tables that the trigger references might be undergoing changes made by SQL statements in other users' transactions. SQL statements running in triggers follow the same rules that standalone SQL statements do. Specifically:

- Queries in the trigger see the current read-consistent materialized view of referenced tables and any data changed in the same transaction.
- Updates in the trigger wait for existing data locks to be released before proceeding.

An **INSTEAD OF** trigger is either:

- A DML trigger created on either a noneditioning view or a nested table column of a noneditioning view
- A system trigger defined on a CREATE statement

The database fires the INSTEAD OF trigger instead of running the triggering statement.

Why use Triggers??

Triggers let user customize the database management system. Triggers can be used to:

- Automatically generate virtual column values
- Log events
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Enforce referential integrity when child and parent tables are on different nodes of a distributed database
- Publish information about database events, user events, and SQL statements to subscribing applications
- Prevent DML operations on a table after regular business hours
- Prevent invalid transactions
- Enforce complex business or referential integrity rules that user cannot define with constraints.
- Triggers are not reliable security mechanisms, because they are programmatic and easy to disable.

Triggers versus Constraints

Both triggers and constraints can constrain data input, but they differ significantly.

1. A trigger always applies to new data only.

   For example, a trigger can prevent a DML statement from inserting a NULL value into a database column, but the column might contain NULL values that were inserted into the column before the trigger was defined or while the trigger was disabled.

   A constraint can apply either to new data only (like a trigger) or to both new and existing data. Constraint behavior depends on constraint state.
2. Constraints are easier to write and less error-prone than triggers that enforce the same rules.
3. However, triggers can enforce some complex business rules that constraints cannot.
4. Oracle strongly recommends that the use of triggers to constrain data input only in these situations:
   - To enforce referential integrity when child and parent tables are on different nodes of a distributed database
   - To enforce complex business or referential integrity rules that user cannot define with constraints.

Classification of Triggers in Oracle

Triggers can be classified based on the following parameters.

- Classification based on the **Timing**
  - BEFORE Trigger: It fires before the specified event has occurred.
  - AFTER Trigger: It fires after the specified event has occurred.
  - INSTEAD OF Trigger: A special type. You will learn more about the further topics. (only for DML)

- Classification based on the **Level**
  - STATEMENT level Trigger: It fires one time for the specified event statement.
  - ROW level Trigger: It fires for each record that got affected in the specified event. [has **FOR EACH ROW** clause] (only for DML)
- Classification based on the **Event**
  - DML Trigger: It fires when the DML event is specified (INSERT/UPDATE /DELETE)
  - DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)
  - DATABASE Trigger: It fires when the database event is specified (LOGON/ LOGOFF/STARTUP/SHUTDOWN)

DML Triggers

A DML trigger is created on either a table or view, and its triggering event is composed of the DML statements DELETE, INSERT, and UPDATE. A DML trigger is either simple or compound.

A compound DML trigger created on a table or editioning view can fire at one, some, or all of the preceding timing points. Compound DML triggers help program an approach where the user want the actions that is implement for the various timing points to share common data. A simple or compound DML trigger that fires at row level can access the data in the row that it is processing.

A simple DML trigger fires at exactly one of these timing points:

- Before the triggering statement runs
  The trigger is called a BEFORE statement trigger or statement-level BEFORE trigger.
- After the triggering statement runs
  The trigger is called an AFTER statement trigger or statement-level AFTER trigger.
- Before each row that the triggering statement affects
  The trigger is called a BEFORE each row trigger or row-level BEFORE trigger.
- After each row that the triggering statement affects
  The trigger is called an AFTER each row trigger or row-level AFTER trigger.

Conditional Predicates for Detecting Triggering DML Statement

The triggering event of a DML trigger can be composed of multiple triggering statements. When one of them fires the trigger, the trigger can determine which one by using these conditional predicates:

Conditional Predicate	**TRUE if and only if:**
`INSERTING`	An INSERT statement fired the trigger.
`UPDATING`	An UPDATE statement fired the trigger.
`UPDATING ('column')`	An UPDATE statement that affected specified column fired the trigger.
`DELETING`	A DELETE statement fired the trigger.

The CREATE TRIGGER syntax -

```
CREATE [ OR REPLACE ] TRIGGER <trigger_name>
      [BEFORE | AFTER | INSTEAD OF ]
      [INSERT | UPDATE | DELETE......]
            ON <name of underlying object>
      [FOR EACH ROW]
      [WHEN<condition for trigger to get execute> ]
DECLARE
      <Declaration part>
BEGIN
      <Execution part>
EXCEPTION
      <Exception handling part>
END;
```

Lets go through an example trigger.

```
CREATE OR REPLACE TRIGGER TRIGGERING_STMT_TEST
      BEFORE
      INSERT OR UPDATE OF QTY, DESCRIPT OR DELETE ON PRODUCT
BEGIN
      CASE
            WHEN INSERTING THEN
                  DBMS_OUTPUT.PUT_LINE('Inserting');
            WHEN UPDATING('QTY') THEN
                  DBMS_OUTPUT.PUT_LINE('Updating Quantity');
            WHEN UPDATING('DESCRIPT') THEN
                  DBMS_OUTPUT.PUT_LINE('Updating Description');
            WHEN DELETING THEN
                  DBMS_OUTPUT.PUT_LINE('Deleting');
      END CASE;
END;
/
```

Check creation of a trigger using the [USER|ALL|DBA]_OBJECTS view and the state information using the [USER|ALL|DBA]_TRIGGERS view.

A statement-level trigger fires **only once** for the triggering event and doesnot have access to column values of each row that is affected by the trigger.

A row-level trigger fires **for each row** that is affected by the trigger and can access the original and new column values processed by the SQL statement.

Correlation Names and Pseudorecords

Applies only to ROW-LEVEL Triggers.

A trigger that fires at row level can access the data in the row that it is processing by using correlation names. The default correlation names are OLD, NEW, and PARENT. To change the correlation names, use the **REFERENCING** clause of the CREATE TRIGGER statement.

If the trigger is created on a nested table in a view, then OLD and NEW refer to the current row of the nested table, and PARENT refers to the current row of the parent table. If the trigger is created on a table or view, then OLD and NEW refer to the current row of the table or view, and PARENT is undefined.

OLD, NEW, and PARENT are also called pseudorecords, because they have record structure, but are allowed in fewer contexts than records are. The structure of a pseudorecord is **table_name%ROWTYPE**, where table_name is the name of the table on which the trigger is created (for OLD and NEW) or the name of the parent table (for PARENT).

OLD and NEW Pseudorecord Field Values

Triggering Statement	**OLD.field Value**	**NEW.field Value**
INSERT	NULL	Post-insert value
UPDATE	Pre-update value	Post-update value
DELETE	Pre-delete value	NULL

The restrictions on pseudorecords are:

- A pseudorecord cannot appear in a record-level operation.

  For example, the trigger cannot include this statement:

  ```
  :NEW := NULL;
  ```
- A pseudorecord cannot be an actual subprogram parameter.

  A pseudorecord field can be an actual subprogram parameter.
- The trigger cannot change OLD field values.

  Trying to do so raises `ORA-04085`.
- If triggering statement is DELETE, then trigger cannot change NEW field values.

  Trying to do so raises `ORA-04084`.
- An AFTER trigger cannot change NEW field values, because the triggering statement runs before the trigger fires.

  Trying to do so raises `ORA-04084`.

A BEFORE trigger can change NEW field values before a triggering INSERT or UPDATE statement puts them in the table.

If a statement triggers both a BEFORE trigger and an AFTER trigger, and the BEFORE trigger changes a NEW field value, then the AFTER trigger "sees" that change.

Referencing Column Values in Trigger [row-level] Body

- For an INSERT statement, the values that will be inserted are contained in :NEW.column_name
- For an UPDATE statement, the original value for the column is contained in :OLD.column_name; the new value for the column is contained in :NEW.column_name
- For a DELETE statement, the column values of row being deleted are contained in :OLD.column_name

Example: A trigger that fires on any triggering event.

```
CREATE TABLE TRG_TEST_TBL (
      EVENT_NAME VARCHAR2(10) DEFAULT 'SomE EvEnT',
      EVENT_TIME TIMESTAMP
);

CREATE OR REPLACE TRIGGER ANY_EVENT_TRIGGER
      AFTER
      INSERT OR UPDATE OR DELETE ON PART
      FOR EACH ROW
BEGIN
      INSERT INTO TRG_TEST_TBL(EVENT_TIME)
            VALUES(SYSTIMESTAMP);
END;
/
```

To test the trigger

```
DELETE FROM PART WHERE 1=1;
ROLLBACK;
SELECT * FROM TRG_TEST_TBL;
```

The BEFORE and AFTER Triggers

A BEFORE row-level triggers is fired before the triggering event is executed. It thus allows user to modify a row's column values. An AFTER row-level trigger fires after the triggering event has occured, thus cannot modify column values.

Possible Triggers for a Table

Based on all permutations in CREATE TRIGGER statement, a single table can have upto 12 types of triggers -

- 06 row-level triggers – BEFORE DELETE, BEFORE INSERT, BEFORE UPDATE, AFTER DELETE, AFTER INSERT, and AFTER UPDATE.
- 06 statement-level triggers – BEFORE DELETE, BEFORE INSERT, BEFORE UPDATE, AFTER DELETE, AFTER INSERT, and AFTER UPDATE.

Example: Create trigger that inserts row in log table after EMPLOYEE.SALARY is updated.

```
CREATE TABLE EMP_SAL_LOG (
      ENO NUMBER(4),
      LOG_DT DATE,
      NEW_SAL NUMBER(7,2),
      ACTION VARCHAR2(25)
);

CREATE OR REPLACE TRIGGER TRG_LOG_SAL_INCR
      AFTER UPDATE OF SALARY ON EMPLOYEE
      FOR EACH ROW
BEGIN
      INSERT INTO EMP_SAL_LOG
            VALUES(:NEW.EID, SYSDATE, :NEW.SALARY, 'NeW Salary..');
END;
/
```

To test the trigger

```
      SELECT EID, FNAME, LNAME, SALARY FROM EMPLOYEE;
      UPDATE EMPLOYEE
            SET SALARY = SALARY * 1.15
            WHERE UPPER(DESIGNATION) = 'PROFESSOR';
      SELECT * FROM EMP_SAL_LOG;
```

Conditional Trigger

In a conditional trigger the database evaluates the WHEN condition for each affected row. If the WHEN condition is TRUE for an affected row, then the trigger fires for that row before the triggering statement runs. If the WHEN condition is not TRUE for an affected row, then trigger does not fire for that row, but the triggering statement still runs.

Example: A conditional trigger that prints salary change information whenever a DML statement affects the EMPLOYEE table — unless that information is about a Professor.

```
CREATE OR REPLACE TRIGGER TRG_SALARY_CHANGES
    BEFORE DELETE OR INSERT OR UPDATE ON EMPLOYEE
    FOR EACH ROW
    WHEN (NEW.DESIGNATION <> 'Professor')  -- exclude Professor
DECLARE
    SAL_DIFF  NUMBER;
BEGIN
    SAL_DIFF := :NEW.SALARY  - :OLD.SALARY;
    DBMS_OUTPUT.PUT(:NEW.EID||': 'Old_Sal = '||:OLD.SALARY
            ||', New_Sal = '||:NEW.SALARY||', Diff = '||SAL_DIFF);
END;
/
```

To test the trigger

```
SELECT EID, FNAME, DEPARTMENT, SALARY, DESIGNATION
      FROM EMPLOYEE
      WHERE DEPARTMENT IN
            ('Biotechnology','Computer Science','Nanotechnology')
      ORDER BY DEPARTMENT, EID;
```

Triggering Action

```
UPDATE EMPLOYEE
      SET SALARY = SALARY * 1.25
      WHERE DEPARTMENT IN
            ('Biotechnology','Computer Science','Nanotechnology');
SELECT EID, FNAME, DEPARTMENT, SALARY, DESIGNATION
      FROM EMPLOYEE
      WHERE DEPARTMENT IN
            ('Biotechnology','Computer Science','Nanotechnology')
      AND DESGINATION = 'Professor';
```

COMMIT and ROLLBACK in Triggers

It is not permitted to execute a COMMIT or a ROLLBACK statement in a database trigger. Also, a trigger may not call a stored procedure, function or a package subprogram that performs a COMMIT or ROLLBACK.

This is due the reason that - if a trigger encounters an error, all database changes that have been propagated by the trigger should be rolled back; but if the trigger committed some portion of the database, the Oracle would not be able to roll back the entire transaction.

Dropping, Enabling and Disabling a Trigger

When a trigger is no longer needed, it may be dropped.

```
DROP TRIGGER trigger_name;
```

Sometimes, dropping a trigger is too drastic. Instead, user may want to deactivate it temporarily. The trigger remains disabled until it is enabled again.

```
ALTER TRIGGER trigger_name DISABLE;
```

A deactivated trigger can be re-activated as below -

```
ALTER TRIGGER trigger_name ENABLE;
```

Cascading Triggers

The interaction of triggers can be quite complex. In usual situations, a trigger when fired may cause another trigger to fire. Triggers that behave in the states manner are called " cascading triggers" .

```
CREATE TABLE TBL1 ( COL1 NUMBER );
CREATE TABLE TBL2 ( COL2 NUMBER );
CREATE TABLE TBL3 ( COL3 NUMBER );
INSERT INTO TBL1 VALUES (7);
INSERT INTO TBL1 VALUES (10);
INSERT INTO TBL1 VALUES (13);


CREATE OR REPLACE TRIGGER TBL1_UPD_B4
      BEFORE UPDATE ON TBL1
      FOR EACH ROW
BEGIN
      INSERT INTO TBL2 VALUES (:OLD.COL1);
END;
/

CREATE OR REPLACE TRIGGER TBL2_INS_B4
      BEFORE INSERT ON TBL2
      FOR EACH ROW
BEGIN
      UPDATE TBL3
            SET COL3 = :NEW.COL2;
END;
/

CREATE OR REPLACE TRIGGER TBL3_UPD_AFTR
      AFTER UPDATE ON TBL3
      FOR EACH ROW
BEGIN
      INSERT INTO TBL3 VALUES (27);
END;
/
```

To test the trigger

```
UPDATE TBL1 SET COL1 = 8;
SELECT * FROM TBL1;
SELECT * FROM TBL2;
SELECT * FROM TBL3;
```

By default, the number of cascaded triggers that can fire is limited to 32.