**Oracle PL/SQL Trigger**

## What are Triggers in PL/SQL?

Triggers are stored programs that are fired automatically when some events occur. The code to be fired can be defined as per the requirement.

Oracle has also provided the facility to mention the event upon which the trigger needs to be fire and the timing of the execution.

**Topics:**

* Benefits of Triggers
* Types of Triggers in Oracle
* How to Create Trigger
* :NEW and :OLD Clause
* INSTEAD OF Trigger
* Compound Trigger

## Benefits of Triggers

Following are the benefits of triggers.

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

## Types 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. (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)

So each trigger is the combination of above parameters.

The syntax for creating a trigger is −

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Where,

* CREATE [OR REPLACE] TRIGGER trigger\_name − Creates or replaces an existing trigger with the *trigger\_name*.
* {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
* [OF col\_name] − This specifies the column name that will be updated.
* [ON table\_name] − This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
* [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

### **Example**

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

dbms\_output.put\_line('Old salary: ' || :OLD.salary);

dbms\_output.put\_line('New salary: ' || :NEW.salary);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

When the above code is executed at the SQL prompt, it produces the following result −

Trigger created.

The following points need to be considered here −

* OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
* The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

## **Triggering a Trigger**

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Kriti', 22, 'HP', 7500.00 );

When a record is created in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result −

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

UPDATE customers

SET salary = salary + 500

WHERE id = 2;

When a record is updated in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result −

Old salary: 1500

New salary: 2000

Salary difference: 500

## :NEW and :OLD Clause

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

* :NEW – It holds a new value for the columns of the base table/view during the trigger execution
* :OLD – It holds old value of the columns of the base table/view during the trigger execution

This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

|  |  |  |  |
| --- | --- | --- | --- |
|  | **INSERT** | **UPDATE** | **DELETE** |
| **:NEW** | VALID | VALID | INVALID. There is no new value in delete case. |
| **:OLD** | INVALID. There is no old value in insert case | VALID | VALID |

## INSTEAD OF Trigger

"INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event is going to occur on the complex view.

Consider an example in which a view is made from 3 base tables. When any DML event is issued over this view, that will become invalid because the data is taken from 3 different tables. So in this INSTEAD OF trigger is used. The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view for the given event.

**Example 1**: In this example, we are going to create a complex view from two base table.

* Table\_1 is emp table and
* Table\_2 is department table.

Then we are going to see how the INSTEAD OF trigger is used to issue UPDATE the location detail statement on this complex view. We are also going to see how the :NEW and :OLD is useful in triggers.

* Step 1: Creating table 'emp' and 'dept' with appropriate columns
* Step 2: Populating the table with sample values
* Step 3: Creating view for the above created table
* Step 4: Update of view before the instead-of trigger
* Step 5: Creation of the instead-of trigger
* Step 6: Update of view after instead-of trigger

**Step 1)** Creating table 'emp' and 'dept' with appropriate columns

CREATE TABLE emp(

emp\_no NUMBER,

emp\_name VARCHAR2(50),

salary NUMBER,

manager VARCHAR2(50),

dept\_no NUMBER);

/

CREATE TABLE dept(

Dept\_no NUMBER,

Dept\_name VARCHAR2(50),

LOCATION VARCHAR2(50));

/

**Code Explanation**

* **Code line 1-7**: Table 'emp' creation.
* **Code line 8-12**: Table 'dept' creation.

**Output**

Table Created

**Step 2)** Now since we have created the table, we will populate this table with sample values and Creation of Views for the above tables.

BEGIN

INSERT INTO DEPT VALUES(10,‘HR’,‘USA’);

INSERT INTO DEPT VALUES(20,'SALES','UK’);

INSERT INTO DEPT VALUES(30,‘FINANCIAL',‘JAPAN');

COMMIT;

END;

/

BEGIN

INSERT INTO EMP VALUES(1000,'XXX5,15000,'AAA',30);

INSERT INTO EMP VALUES(1001,‘YYY5,18000,‘AAA’,20) ;

INSERT INTO EMP VALUES(1002,‘ZZZ5,20000,‘AAA',10);

COMMIT;

END;

/

**Code Explanation**

* **Code line 13-19**: Inserting data into 'dept' table.
* **Code line 20-26:**Inserting data into 'emp' table.

**Output**

PL/SQL procedure completed

**Step 3)**Creating a view for the above created table.

CREATE VIEW tech\_emp\_view(

Employee\_name:dept\_name,location) AS

SELECT emp.emp\_name,dept.dept\_name,dept.location

FROM emp,dept

WHERE emp.dept\_no=dept.dept\_no;

/

SELECT \* FROM tech\_emp\_view;

**Code Explanation**

* **Code line 27-32:**Creation of 'tech\_emp\_view' view.
* **Code line 33:** Querying tech\_emp\_view.

**Output**

View created

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_NAME** | **DEPT\_NAME** | **LOCATION** |
| ZZZ | HR | USA |
| YYY | SALES | UK |
| XXX | FINANCIAL | JAPAN |

**Step 4)** Update of view before instead-of trigger.

BEGIN

UPDATE tech\_emp\_view SET location='FRANCE' WHERE employee\_name=:'XXX’;

COMMIT;

END;

/

**Code Explanation**

* **Code line 34-38:**Update the location of "XXX" to 'FRANCE'. It raised the exception because the DML statements are not allowed in the complex view.

**Output**

ORA-01779: cannot modify a column which maps to a non key-preserved table

ORA-06512: at line 2

**Step 5)**To avoid the error encounter during updating view in the previous step, in this step we are going to use "instead of trigger."

CREATE TRIGGER tech\_view\_modify\_trg

INSTEAD OF UPDATE

ON tech\_emp\_view

FOR EACH ROW

BEGIN

UPDATE dept

SET location=:new.location

WHERE dept\_name=:old.dept\_name;

END;

/

**Code Explanation**

* **Code line 39:** Creation of INSTEAD OF trigger for 'UPDATE' event on the 'tech\_emp\_view' view at the ROW level. It contains the update statement to update the location in the base table 'dept'.
* **Code line 44:** Update statement uses ':NEW' and ': OLD' to find the value of columns before and after the update.

**Output**

Trigger Created

**Step 6)** Update of view after instead-of trigger. Now the error will not come as the "instead of trigger" will handle the update operation of this complex view. And when the code has executed the location of employee XXX will be updated to "France" from "Japan."

BEGIN

UPDATE tech\_emp\_view SET location='FRANCE' WHERE employee\_name='XXX';

COMMIT;

END;

/

SELECT \* FROM tech\_emp\_view;

**Code Explanation:**

* **Code line 49-53:** Update of the location of "XXX" to 'FRANCE'. It is successful because the 'INSTEAD OF' trigger has stopped the actual update statement on view and performed the base table update.
* **Code line 55:** Verifying the updated record.

**Output:**

PL/SQL procedure successfully completed

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_NAME** | **DEPT\_NAME** | **LOCATION** |
| ZZZ | HR | USA |
| YYY | SALES | UK |
| XXX | FINANCIAL | FRANCE |

## Compound Trigger

The Compound trigger is a trigger that allows you to specify actions for each of four timing points in the single trigger body. The four different timing point it supports is as below.

* BEFORE STATEMENT – level
* BEFORE ROW – level
* AFTER ROW - level
* AFTER STATEMENT – level

It provides the facility to combine the actions for different timing into the same trigger.

CREATE [ OR REPLACE ] TRIGGER <trigger\_name>

FOR

[INSERT | UPDATE | DELET.......]

ON <name of underlying object>

<Declarative part>

BEFORE STATEMENT IS

BEGIN

<Execution part>;

END BEFORE STATEMENT;

BEFORE EACH ROW IS

BEGIN

<Execution part>;

END EACH ROW;

AFTER EACH ROW IS

BEGIN

<Execution part>;

END AFTER EACH ROW;

AFTER STATEMENT IS

BEGIN

<Execution part>;

END AFTER STATEMENT;

END;

**Syntax Explanation:**

* The above syntax shows the creation of 'COMPOUND' trigger.
* Declarative section is common for all the execution block in the trigger body.
* These 4 timing blocks can be in any sequence. It is not mandatory to have all these 4 timing blocks. We can create a COMPOUND trigger only for the timings which are required.

**Example 1**: In this example, we are going to create a trigger to auto-populate the salary column with the default value 5000.

CREATE TRIGGER emp\_trig

FOR INSERT

ON emp

COMPOUND TRIGGER

BEFORE EACH ROW IS

BEGIN

:new.salary:=5000;

END BEFORE EACH ROW;

END emp\_trig;

/

BEGIN

INSERT INTO EMP VALUES(1004,‘CCC’,15000,‘AAA’,30);

COMMIT;

END;

/

SELECT \* FROM emp WHERE emp\_no=1004;

**Code Explanation:**

* **Code line 2-10**: Creation of compound trigger. It is created for timing BEFORE ROW- level to populate the salary with default value 5000. This will change the salary to default value '5000' before inserting the record into the table.
* **Code line 11-14**: Insert the record into 'emp' table.
* **Code line 16**: Verifying the inserted record.

**Output:**

Trigger created

PL/SQL procedure successfully completed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **EMP\_NO** | **SALARY** | **MANAGER** | **DEPT\_NO** |
| CCC | 1004 | 5000 | AAA | 30 |

**Enabling and Disabling Triggers**

Triggers can be enabled or disabled. To enable or disable the trigger, an ALTER (DDL) statement needs to be given for the trigger that disable or enable it.

Below are the syntax for enabling/disabling the triggers.

ALTER TRIGGER <trigger\_name> [ENABLE|DISABLE];

ALTER TABLE <table\_name> [ENABLE|DISABLE] ALL TRIGGERS;

**Syntax Explanation:**

* The first syntax shows how to enable/disable the single trigger.
* The second statement shows how to enable/disable all the triggers on a particular table.

### **Triggers Example**

## **Inserting Trigger**

This trigger execute BEFORE to convert ename field lowercase to uppercase.

CREATE or REPLACE TRIGGER trg1

BEFORE

INSERT ON emp1

FOR EACH ROW

BEGIN

:new.ename := upper(:new.ename);

END;

/

## **Restriction to Deleting Trigger**

This trigger is preventing to deleting row.

Delete Trigger Example

CREATE or REPLACE TRIGGER trg1

AFTER

DELETE ON emp1

FOR EACH ROW

BEGIN

IF :old.eno = 1 THEN

raise\_application\_error(-20015, 'You can't delete this row');

END IF;

END;

/

## **Example**

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

**Create table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create trigger:**

Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

1. **CREATE** OR REPLACE **TRIGGER** display\_salary\_changes
2. BEFORE **DELETE** OR **INSERT** OR **UPDATE** **ON** customers
3. **FOR** EACH ROW
4. **WHEN** (NEW.ID > 0)
5. **DECLARE**
6. sal\_diff number;
7. **BEGIN**
8. sal\_diff := :NEW.salary  - :OLD.salary;
9. dbms\_output.put\_line('Old salary: ' || :OLD.salary);
10. dbms\_output.put\_line('New salary: ' || :NEW.salary);
11. dbms\_output.put\_line('Salary difference: ' || sal\_diff);
12. **END**;
13. /

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

**Check the salary difference by procedure:**

Use the following code to get the old salary, new salary and salary difference after the trigger created.

1. **DECLARE**
2. total\_rows number(2);
3. **BEGIN**
4. **UPDATE**  customers
5. **SET** salary = salary + 5000;
6. IF sql%notfound **THEN**
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found **THEN**
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. **END** IF;
12. **END**;
13. /

Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

**Note:** As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

## **Important Points**

Following are the two very important point and should be noted carefully.

* OLD and NEW references are used for record level triggers these are not avialable for table level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

## **Trigger Execution Hierarchy**

The following hierarchy is followed when a trigger is fired.  
**1)**BEFORE statement trigger fires first.  
**2)** Next BEFORE row level trigger fires, once for each row affected.   
**3)** Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.  
**4)** Finally the AFTER statement level trigger fires.

**For Example:** Let's create a table 'product\_check' which we can use to store messages when triggers are fired.

*CREATE TABLE product*

*(Message varchar2(50),*

*Current\_Date number(32)*

*);*

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

**1) BEFORE UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' before a sql update statement is executed, at the statement level.

*CREATE or REPLACE TRIGGER Before\_Update\_Stat\_product*

*BEFORE*

*UPDATE ON product*

*Begin*

*INSERT INTO product\_check*

*Values('Before update, statement level',sysdate);*

*END;*

*/*

**2) BEFORE UPDATE, Row Level:**This trigger will insert a record into the table 'product\_check' before each row is updated.

*CREATE or REPLACE TRIGGER Before\_Upddate\_Row\_product*

*BEFORE*

*UPDATE ON product*

*FOR EACH ROW*

*BEGIN*

*INSERT INTO product\_check*

*Values('Before update row level',sysdate);*

*END;*

*/*

**3) AFTER UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' after a sql update statement is executed, at the statement level.

*CREATE or REPLACE TRIGGER After\_Update\_Stat\_product*

*AFTER*

*UPDATE ON product*

*BEGIN*

*INSERT INTO product\_check*

*Values('After update, statement level', sysdate);*

*End;*

*/*

**4) AFTER UPDATE, Row Level:**This trigger will insert a record into the table 'product\_check' after each row is updated.

*CREATE or REPLACE TRIGGER After\_Update\_Row\_product*

*AFTER*

*insert On product*

*FOR EACH ROW*

*BEGIN*

*INSERT INTO product\_check*

*Values('After update, Row level',sysdate);*

*END;*

*/*

Now lets execute a update statement on table product.

*UPDATE PRODUCT SET unit\_price = 800*

*WHERE product\_id in (100,101);*

Lets check the data in 'product\_check' table to see the order in which the trigger is fired.

*SELECT \* FROM product\_check;*

**Output:**

Mesage                                             Current\_Date

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

Before update, statement level          26-Nov-2008  
Before update, row level                    26-Nov-2008  
After update, Row level                     26-Nov-2008  
Before update, row level                    26-Nov-2008  
After update, Row level                     26-Nov-2008  
After update, statement level            26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

## **How To know Information about Triggers.**

We can use the data dictionary view 'USER\_TRIGGERS' to obtain information about any trigger.

The below statement shows the structure of the view 'USER\_TRIGGERS'

*DESC USER\_TRIGGERS;*

NAME                              Type

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

TRIGGER\_NAME                 VARCHAR2(30)  
TRIGGER\_TYPE                  VARCHAR2(16)  
TRIGGER\_EVENT                VARCHAR2(75)   
TABLE\_OWNER                  VARCHAR2(30)  
BASE\_OBJECT\_TYPE           VARCHAR2(16)  
TABLE\_NAME                     VARCHAR2(30)  
COLUMN\_NAME                  VARCHAR2(4000)  
REFERENCING\_NAMES        VARCHAR2(128)  
WHEN\_CLAUSE                  VARCHAR2(4000)  
STATUS                            VARCHAR2(8)  
DESCRIPTION                    VARCHAR2(4000)  
ACTION\_TYPE                   VARCHAR2(11)  
TRIGGER\_BODY                 LONG

This view stores information about header and body of the trigger.

*SELECT \* FROM user\_triggers WHERE trigger\_name = 'Before\_Update\_Stat\_product';*

The above sql query provides the header and body of the trigger 'Before\_Update\_Stat\_product'.

You can drop a trigger using the following command.

*DROP TRIGGER trigger\_name;*

## **CYCLIC CASCADING in a TRIGGER**

This is an undesirable situation where more than one trigger enter into an infinite loop. while creating a trigger we should ensure the such a situtation does not exist.

The below example shows how Trigger's can enter into cyclic cascading.  
Let's consider we have two tables 'abc' and 'xyz'. Two triggers are created.  
**1)** The INSERT Trigger, triggerA on table 'abc' issues an UPDATE on table 'xyz'.  
**2)**The UPDATE Trigger, triggerB on table 'xyz' issues an INSERT on table 'abc'.

In such a situation, when there is a row inserted in table 'abc', triggerA fires and will update table 'xyz'.   
When the table 'xyz' is updated, triggerB fires and will insert a row in table 'abc'.  
This cyclic situation continues and will enter into a infinite loop, which will crash the database.