Trigger

- Triggers are procedures that are stored in the database and implicitly run, or **fired**, when something happens.
- It is PL/SQL block or a PL/SQL procedure, C, or Java procedure associated with a table, view, schema or the database.
- Oracle automatically executes a trigger when a specified event takes place, which may be in the form of a system event or a DML/DDL statement being issued against the table.
- Two Types :
  Application trigger : Fires whenever an event occurs with a particular application
  Database trigger : Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database.

Uses of Triggers

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Database trigger creation syntax (we will cover only the database trigger)

```
CREATE [OR REPLACE ] TRIGGER trigger_name
    {BEFORE | AFTER | INSTEAD OF }
    {INSERT [OR] | UPDATE [OR] | DELETE}
    [OF col_name]
    ON table_name                          } Triggering event

    [REFERENCING OLD AS o NEW AS n]
    [FOR EACH ROW]
    WHEN (condition)                       } Trigger constraint
    DECLARE
        Declaration-statements
    BEGIN
        Executable-statements
        EXCEPTION
        Exception-handling-statements      } Trigger body /action
    END;
```

1. A triggering event - insert, update, delete or a instance shutdown or startup etc.
2. A trigger constraint (Optional) - specifies a Bollean expression that must be true for the trigger to fire. This condition is specified using the WHEN clause.
3. Trigger action - contains PL/SQl code to be executed when the trigger fires.

Explanation of different clauses in syntax

- *CREATE [OR REPLACE] TRIGGER trigger_name* - creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE | AFTER | INSTEAD OF }* - indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view (before and after cannot be used to create a trigger on a view).
- *{INSERT [OR] | UPDATE [OR] | DELETE}* - More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col_name]* - used with **update** triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *[ON table_name]* - identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - used to reference the old and new values of the data being changed. By default, you reference the values as **:old.column_name** or **:new.column_name**. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- *[FOR EACH ROW]* - used to determine whether a trigger must fire when each row gets affected (i.e. a **Row Level Trigger**) or just once when the entire sql statement is executed (i.e. **statement** level Trigger).
- *WHEN (condition)* - valid only for **row level triggers**. The trigger is fired only for rows that satisfy the condition specified.

Points to remember

- The body of a trigger can contain DML SQL statements. It can also contain SELECT statements, but they must be SELECT... INTO... statements or the SELECT statement in the definition of a cursor.
- DDL statements are not allowed in the body of a trigger.
- A trigger cannot include transaction control statements like COMMIT, SAVEPOINT and ROLLBACK.
- A table can have any number of triggers.
- A trigger is automatically enabled when it is created. To enable or disable a trigger using **ALTER TRIGGER** command

To enable the disabled trigger named REORDER of the INVENTORY table, enter the statement: **ALTER TRIGGER Reorder ENABLE;** For disable **: ALTER TRIGGER Reorder DISABLE;**	For example, to enable all triggers defined for a specific table, enter the following statement: **ALTER TABLE Inventory ENABLE ALL TRIGGERS;** For disable : **ALTER TABLE Inventory DISABLE ALL TRIGGERS;**

PL/SQL 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. The triggers 2 and 3 will be fired for each affected row.
4. Finally the AFTER statement level trigger fires.

To check the hierarchy of execution, we create a **product_check** table which we can use to store messages when triggers are fired. Then we create following triggers and check the records inserted in the product_check table.

CREATE TABLE product_check (Message varchar2(50), Current_Date number(32));

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

<table>
<tr>
<td>

- BEFORE UPDATE, Statement Level: 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;
/
```

</td>
<td>

- BEFORE UPDATE, Row Level: 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;
/
```

</td>
</tr>
<tr>
<td>

- AFTER UPDATE, Statement Level: 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
```

</td>
<td>

- AFTER UPDATE, Row Level: insert a record into the table 'product_check' after each row is updated.

```
CREATE or REPLACE TRIGGER
                After_Update_Row_product
AFTER  UPDATE On product
FOR EACH ROW
BEGIN
```

</td>
</tr>
</table>

```
    INSERT INTO product_check
    Values('After update, statement level',
sysdate);
 End;
 /
```

```
  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;

```
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
```

- 'before update' and 'after update' row level events have occurred 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.

Cycle Cascading in Trigger

This is an undesirable situation where more than one triggers enter into an infinite loop. While creating a trigger we should ensure that such a situation does not exist. The follow 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 an infinite loop, which will crash the database.

Trigger Example

1. The following statement creates a trigger for the Emp_tab table.

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE UPDATE ON Emp_tab
  FOR EACH ROW
WHEN (new.Empno > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff  := :new.sal  - :old.sal;
   dbms_output.put('Old salary: ' || :old.sal);
   dbms_output.put('  New salary: ' || :new.sal);
   dbms_output.put_line('  Difference ' || sal_diff);
END;
/
```

Indicates a row level trigger

The trigger is fired when DML operations (UPDATE statements) is performed on the table. You can choose what combination of operations should fire the trigger.

Because the trigger uses the FOR EACH ROW clause, it might be executed multiple times, such as when updating or deleting multiple rows. You might omit this clause if you just want to record the fact that the operation occurred, but not examine the data for each row.

Once the trigger is created, entering the following SQL statement:

UPDATE Emp_tab SET sal = sal + 500.00 WHERE deptno = 10;

fires once for each row that is updated, in each case printing the new salary, old salary, and the difference.

- The CREATE (or CREATE OR REPLACE) statement fails if any errors exist in the PL/SQL block.
- The **size** of the trigger cannot be more **than 32K.**

2. Creating auto increment column in Oracle

There is no such thing as "auto_increment" or "identity" columns in **Oracle *as of Oracle 11g***. However, you can model it easily with a sequence and a trigger:

Table definition:

```
CREATE TABLE departments (
  ID                NUMBER(10)    NOT NULL,
  DESCRIPTION  VARCHAR2(50)  NOT NULL);

ALTER TABLE departments ADD (
  CONSTRAINT dept_pk PRIMARY KEY (ID));
```

CREATE SEQUENCE dept_seq START WITH 1;

Trigger definition:

```
CREATE OR REPLACE TRIGGER dept_bir
BEFORE INSERT ON departments
FOR EACH ROW
BEGIN
  SELECT dept_seq.NEXTVAL
  INTO   :new.id
  FROM   dual;
END;
/
```

When we are using SQLDeveloper tool we can specify the identity column and it will automatically add one sequence and one trigger to implement the identity column.

IDENTITY column is now available on Oracle 12c:

We can use any of the following syntax to generated identity column in Oracle 12c It is similar to the **AUTO_INCREMENT column** in MySQL or **IDENTITY column** in SQL Server.

```
create table t1 (
  c1 NUMBER GENERATED by default on null as IDENTITY,
  c2 VARCHAR2(10)
  );
```

- GENERATED keyword is mandatory.
- GENERATED ALWAYS means Oracle always generates a value for the identity column. Attempt to insert a value into the identity column will cause an error.
- GENERATED BY DEFAULT: Oracle generates a value for the identity column if you provide no value. If you provide a value, Oracle will insert that value into the identity column. For this option, Oracle will issue an error if you insert a NULL value into the identity column.
- GENERATED BY DEFAULT ON NULL: Oracle generates a value for the identity column if you provide a NULL value or no value at all.

```
create table t1 (
  c1 NUMBER GENERATED ALWAYS as IDENTITY(START with 1 INCREMENT by 1),
  c2 VARCHAR2(10)
  );
```

3. The price of a product changes constantly. It is important to maintain the history of the prices of the products. We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

- Create two tables as per following create SQL:

CREATE TABLE **product_price_history** (product_id number(5), product_name varchar2(32), supplier_name varchar2(32), unit_price number(7,2));	CREATE TABLE **product** (product_id number(5), product_name varchar2(32), supplier_name varchar2(32), unit_price number(7,2));

- Update the price of a product.

UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

```
CREATE OR REPLACE TRIGGER Update_Price_History
AFTER  UPDATE ON product
FOR EACH ROW
BEGIN
   INSERT INTO product_price_history
   Values(:old.product_id,  :old.product_name,  old:supplier_name, old:unit_price) ;
End;
/
```

If you ROLLBACK the transaction before committing to the database, the data inserted to the table (product_price_history) **is also rolled back.**

- **We assume that id column is an identity column which is automatically incremented by a sequence defined for this table.**

4. **Validation for system date** : SYSDATE cannot be used in CHECK constraint. If we use it will give an error

```
ALTER  TABLE  t1  ADD  CONSTRAINT  chk_Date1
        CHECK (  date1 > current_date);     or    CHECK ( date1 > SYSDATE)
```

Error : ORA : 02346 error
This type of validation need to be handled in trigger

```
CREATE  OR REPLACE TRIGGEr  tg_date    INSERT OR UPDATE
ON  t1   FOR  EACH  ROW
BEGIN
    if  :new.date1  <  sysdate  then
        RAISE_APPLICATION_ERROR ( -200,  ' Date is not less than current date ')
    End if ;
END;
/
```

5. Trigger for Complex Check Constraints:

This trigger performs a complex check before allowing the triggering statement to run.

Create a table : **CREATE TABLE Salgrade** (Grade NUMBER, Losal NUMBER, Hisal NUMBER, Job_classification NUMBER)

```
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT OR UPDATE OF Sal, Job ON Emp99
FOR EACH ROW
```

```
DECLARE
  Minsal            NUMBER;
  Maxsal            NUMBER;
  Salary_out_of_range   EXCEPTION;
BEGIN

   /* Retrieve the minimum and maximum salary for the  employee's new job classification from the
      SALGRADE  table into MINSAL and MAXSAL: */

   SELECT Minsal, Maxsal INTO Minsal, Maxsal FROM Salgrade   WHERE Job_classification = :new.Job;

   /* If the employee's new salary is less than or greater  than the job classification's limits, the
      exception is raised.  The exception message is returned and the  pending INSERT or UPDATE
     statement that fired the trigger is rolled back:*/

     IF (:new.Sal < Minsal OR :new.Sal > Maxsal) THEN
          RAISE Salary_out_of_range;
     END IF;
     EXCEPTION
        WHEN Salary_out_of_range THEN
            Raise_application_error (-20300,  'Salary '||TO_CHAR(:new.Sal)||' out of range for '
                ||'job classification '||:new.Job ||' for employee '||:new.Ename);
        WHEN NO_DATA_FOUND THEN
            Raise_application_error(-20322, 'Invalid Job Classification ' ||:new.Job_classification);
END;
```

We will not consider the instead of Trigger here.

Assertion (English meaning) : a confident and forceful statement of fact or belief.

Database Assertions

Most relational database management systems (RDBMS) do not implement assertions.

- An assertion is a **named constraint** that may relate to the content of individual rows of a table, to the entire contents of a table, or to a state required to exist among a number of tables.
- An assertion is described by an assertion descriptor. In addition it includes: the <search condition>.
- An assertion is satisfied if and only if the specified <searchcondition> is true.

- Assertions are similar to **check** constraints, but unlike check constraints they are not defined on table or column level but are defined on **schema level**. (i.e., assertions are **database objects** of their own right and are not defined within a create table or alter table statement.)

- In SQL, users can specify general constraints via **declarative assertions,** using the **CREATE ASSERTION** statement of the DDL. Those do not fall into table or column level

  CREATE ASSERTION <constraint name> CHECK (<search condition>)

- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- The <searchcondition> , the SQL expression should be always true. When created, the expression must be true.
- DBMS checks the assertion after **any change** that may violate the expression
- Assertions are not linked to specific tables in the database and not linked to specific events whereas Triggers are linked to specific tables and specific events

Example

To specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for" in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS (SELECT *  FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
                    WHERE E.SALARY > M.SALARY AND E.DNO= D.DNUMBER AND
                          D.MGRSSN=M.SSN) );
```

- The constraint name SALARY_CONSTRAINT is followed by the keyword **CHECK** , which is followed by a **Condition** in parentheses that must hold true on every database state for the assertion to be satisfied.
- The constraint name can be used later to refer to the constraint or to modify or drop it.
- The DBMS is responsible for ensuring that the condition is not violated.
- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE , the constraint is violated.

- By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty. Thus, the assertion is violated if the result of the query is not empty. In our example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

- Difference between other CHECK constraint and CREATE ASSERTION : assertions are checked only when UPDATE or INSERT actions are performed against the table.

- Difference between CREATE ASSERTION and CREATE TRIGGER :

Trigger	Assertion
Triggers are more powerful because they can check conditions and also modify the data.	Assertions do not modify the data, they only check certain conditions.
Triggers are linked to specific tables and specific events.	Assertions are not linked to specific tables in the database and not linked to specific events.

For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to monitor the database.

Other actions may be specified, such as executing a specific stored procedure or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL. A trigger specifies an event (such as a particular database update operation), a condition, and an action. The action is to be executed automatically if the condition is satisfied when the event occurs.

Example : Sum of loans taken by a customer does not exceed 100,000

```
Create Assertion SumLoans Check
(   100,000 >= ALL
    Select   Sum(amount)
    From     borrower B , loan L
    Where  B.loan_number = L.loan_number
    Group By customer_name );
```

Example : Number of accounts for each customer in a given branch is at most two

```
Create Assertion NumAccounts Check
(   2 >= ALL
    Select   count(*)
    From     account A , depositor D
    Where  A.account_number =
             D.account_number
    Group By customer_name, branch_name );
```

Example: Average GPA is > 3.0 and average sizeHS is < 1000

```
CREATE ASSERTION Avgs CHECK
  ( 3.0 < (SELECT avg(GPA) FROM Student)
      AND 1000 > (SELECT avg(sizeHS)
      FROM Student))
```

Example: A student with GPA < 3.0 can only apply to campuses with rank > 4.

```
CREATE ASSERTION RestrictApps CHECK(NOT EXISTS
(SELECT * FROM Student, Apply, Campus
        WHERE Student.ID = Apply.ID
        AND Apply.location = Campus.location
        AND Student.GPA < 3.0
        AND Campus.rank <= 4))
```

Note : ALL - The ALL comparison condition is used to compare a value to a list or subquery. It must be preceded by =, !=, >, <, <=, >= and followed by a list or subquery.