RDMS-04-Triggers

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match.
Triggers are stored programs, which are automatically executed or fired when some event occurs.
Triggers could be defined on the table, view, schema, or database with which the event is associated.
Triggers are written to be executed in response to any of the following events:

- A database manipulation (DML) statement (`DELETE`, `INSERT`, or `UPDATE`).
- A database definition (DDL) statement (`CREATE`, `ALTER`, or `DROP`).
- A database operation (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`).

Features of Triggers

- Triggers are fired in response to specific **events** that occur on a table or view.
- These events include:
  - **BEFORE INSERT**: Triggered before an insert operation is performed on a table
  - **AFTER INSERT**: Triggered after an insert operation is performed on a table.
  - **BEFORE UPDATE**: Triggered before an update operation is performed on a table.
  - **AFTER UPDATE**: Triggered after an update operation is performed on a table.
  - **BEFORE DELETE**: Triggered before a delete operation is performed on a table.
  - **AFTER DELETE**: Triggered after a delete operation is performed on a table.
  - **Timing**: Triggers can be defined to execute either before or after the triggering event occurs.
- **Scope**: Triggers can be defined at the statement level or row level.
- **Statement-level Triggers**: Execute once for each triggering event, regardless of the number of rows affected.
- **Row-level Triggers**: Execute for each row affected by the triggering event.
- **Access to Old and New Values**: Triggers can access both old and new values of the data being modified.
- **:OLD** values represent the original values of the row before the trigger event.
- **:NEW** values represent the new values of the row after the trigger event.
- **Conditions**: Triggers can have conditions specified using WHEN clause, which determines whether the trigger should be executed or not based on certain criteria.
- **Transition Variables**: These are used in row-level triggers to manage the state of a row during its life cycle.
  - `INSERTING`: A row is being inserted.
  - `UPDATING`: A row is being updated.
  - `DELETING`: A row is being deleted.

Advantages 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

Syntax

```
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;
```

1. CREATE [OR REPLACE] TRIGGER name_of_trigger:

- CREATE: This starts the creation of a trigger.
- OR REPLACE: This is an optional clause that allows updating an existing trigger with a similar name.
- TRIGGER trigger_name: This mentions the name of the trigger being created or updated.

2. {BEFORE | AFTER | INSTEAD OF}:

- BEFORE: This means that the trigger gets executed before triggering event
- AFTER: This means that the trigger gets executed after triggering event.
- INSTEAD OF: This is used for triggers on views, allowing us to replace default action with the trigger's defined custom action.

3. {INSERT [OR] | UPDATE [OR] | DELETE}:

- This clause mentions the type of DML operation that the trigger will respond to.
- These are the available options:
  - INSERT: The trigger executes in response to an INSERT statement.
  - UPDATE: The trigger executes in response to an UPDATE statement.
  - DELETE: The trigger executes in response to a DELETE statement.

4. [OF col_name]:

- This clause is an **optional** clause mentioning the name of column affected by the trigger.

5. [ON table_name]:

- This clause is an **optional** clause that associates trigger with a specific table.

6. [REFERENCING OLD AS o NEW AS n]:

- This clause is an **optional** clause allowing the references to old and new values in the trigger body.

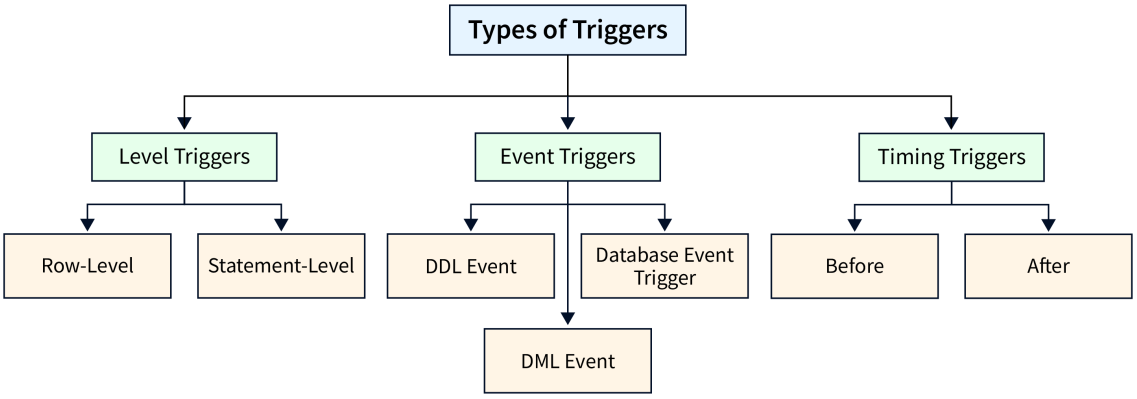
7. [FOR EACH ROW]:

- This is an **optional** clause indicating a row-level trigger.
- This gets executed once for each affected row and without this clause, the trigger is considered a statement-level trigger, executing once independent of number of affected rows.

8. WHEN (condition):

- This is an **optional** clause specifying a condition that, triggers the action defined in the trigger body when true.

Types of Triggers



Categorization on Trigger-Level

- **ROW Level trigger**: It gets executed for each record that got updated by a DML statement.
- **STATEMENT Level trigger**: It gets executed only once by the event statement.

Categorization of the Trigger-Event

- **DML trigger**: It gets executed if a DML event like an UPDATE, INSERT, or DELETE is performed.
- **DDL trigger**: It gets executed if a DDL event like a DROP, ALTER, or CREATE is performed.
- **DATABASE trigger**: It gets executed if a database event like SHUTDOWN, STARTUP, LOGOFF, and LOGON has taken place.

Categorization of the Trigger-Timing

- **BEFORE trigger**: It gets executed prior to the specific event that has taken place.
- **AFTER trigger**: It gets executed post the specific event that has taken place.
- **INSTEAD OF trigger**: It is a special type of trigger and it gets executed for each record that got updated by a DML statement.

Examples

Example on After Insert Trigger: The AFTER INSERT Triggers will fire after the completion of the Insert operation.

Create a trigger to insert details of employee entry made, in EmpLog table.

```
-- STEP:01 TABLE

CREATE TABLE Employees(
  Eid INT PRIMARY KEY,
  Ename varchar(20) NOTNULL,
  City varchar(20),
  Salary number(8, 2) NOT NULL
);

CREATE TABLE EmpLog (
  Eid INT PRIMARY KEY,
  Ename varchar(20) NOTNULL,
  City varchar(20),
  Salary number(8, 2) NOT NULL,
  Uname varchar(20),
  EntryDate date
);

-- STEP:02 TRIGGER

create or replace Trigger After_Insert_Emp
After Insert on
Employees
For Each Row
    Declare
    Begin
        Insert into Emplog
        values (:New.eid, :New.ename, :New.city, :New.salary,(Select userfrom dual),
        dbms_output.put_line('After Insert trigger Successfully executed');
    End;
```

- **Test Trigger:**

1. INSERT INTO Employees values(1, 'John', 'London', 25000);
2. SELECT * FROM Employees

EID	ENAME	CITY	SALARY
1	John	London	25000

3. SELECT * FROM EmpLog

EID	ENAME	CITY	SALARY	UNAME	ENTRYDATE
1	John	London	25000	JOHN123	20-Apr-24

Example on After Update Trigger: The AFTER Update Triggers will fire after the completion of the After update operation.

Consider the below given tables. Create a trigger to update details of employee entry made, in EmpUpdLog table

```
-- STEP:01 TABLES

CREATE TABLE Employees(
  Eid INT PRIMARY KEY,
  Ename varchar(20) NOTNULL,
  City varchar(20),
  Salary number(8, 2) NOT NULL
);

CREATE TABLE EmpUpdLog (
  Eid INT PRIMARY KEY,
  Ename varchar(20) NOT NULL,
  OldCity varchar(20),
  NewCity varchar(20),
  oldSalary number(8, 2) NOT NULL,
  NewSalary number(8, 2) NOT NULL,
  Uname varchar(20),
  UpdateDate date
);

-- STEP:02 TRIGGERS
create or replace Trigger After_Update_Emp
After Update on
Employees
For Each Row
    Declare
    Begin
        Insert into EmpUpdLog
        values (:New.eid, :New.ename, :old.city, :New.city, :old.salary, :New.salary
        dbms_output.put_line('After Update trigger Successfully executed');
    End;
```

- **Test Triggers:**

1. Update record of employee: `UPDATE Employees SET Salary=30000 WHERE Eid=1;`
2. Now check entry in EmpUpdLog for new as well as old data: `SELECT * FROM EmpUpdLog`

EID	ENAME	OLDCITY	NEWCITY	OLDSALARY	NEWSALARY	UNAME	ENTRYDATE
1	John	London	London	25000	30000	JOHN123	20-Apr-24

Example on After Delete Trigger: The AFTER Delete Triggers will fire after the completion of the After delete operation.

Consider the below given tables. Create a trigger to delete details of employee entry made, in EmpDelLog table.

```
-- STEP:01 TABLE

CREATE TABLE Employees(
  Eid INT PRIMARY KEY,
  Ename varchar(20) NOTNULL,
  City varchar(20),
  Salary number(8, 2) NOT NULL
);

CREATE TABLE EmpDelLog (
  Eid INT PRIMARY KEY,
  Ename varchar(20) NOTNULL,
  City varchar(20),
  Salary number(8, 2) NOT NULL,
  uname varchar(20),
  DelDate date
);

-- STEP:02 TRIGGER

create or replace trigger Trig_After_Del
After delete on
employees
for each row
    declare
    begin
        insert into EmpDelLog
        values (:old.Eid, :old.Ename, :old.city, :old.salary, (Select user from dual
        dbms_output.put_line('After Delete trigger Successfully executed');
    end;
```

- **Test Trigger:**

1. Delete record from employees table: `DELETE FROM Employees`
   After Delete trigger Successfully executed
2. All records those are deleted are entered in Empdellog table: `SELECT * FROM EmpDelLog`

EID	ENAME	CITY	SALARY	UNAME	DELDATE
1	John	London	25000	JOHN123	21-Apr-24

Example on Before Insert Trigger: A BEFORE INSERT Trigger means that Oracle will fire this trigger before the INSERT operation is executed

Create table given below and calculate percentage, total and grade before inserting record.

```
-- STEP:01 TABLE
Create Table Stu_Table (
  Stu_Id int Primary key,
  Stu_Name Varchar(15),
  Sub1 int,
  Sub2 int,
  Sub3 int,
  Sub4 int,
  Sub5 int,
  total int,
  per float,
  status varchar(15)
);

-- STEP:02 TRIGGER

CREATE orreplace TRIGGER Trig_Bef_Insert Before
Insert ON
stu_table
FOR EACH ROW BEGIN
    :new.total:= :new.sub1 + :new.sub2 + :new.sub3 +:new.sub4 +:new.sub5;
:new.per:= :new.total / 5;

if : new.per ⩾ 70 then : new.status := 'Dist';
elsif : new.per ⩾ 60
and : new.per < 70 then : new.status := 'First';
elsif : new.per ⩾ 50
and : new.per < 60 then : new.status := 'Second';
elsif : new.per ⩾ 40
and : new.per < 50 then : new.status := 'Third';
else : new.status := 'Pass class';
end if;
dbms_output.put_line('Before Update Trigger Executed');
END;
```

- **Test Trigger:**

1. Inserting data in Table:

```
INSERT INTO Stu_Table (stu_id, stu_name, sub1, sub2, sub3, sub4, sub5)
values
  (1, 'John', 89, 76, 72, 80, 78)
```

2. Now check Stu_Table: `SELECT * FROM Stu_Table`

STU_ID	STU_NAME	SUB1	SUB2	SUB3	SUB4	SUB5
1	John	89	76	72	80	78

Before Update Trigger: A BEFORE UPDATE Trigger means that Oracle will fire this trigger before the UPDATE operation is executed.

Create table given below and create a trigger that checks, if updated account balance is negative, trigger change that to 0.

```
-- STEP:01 TABLE
CREATE TABLE cust_accounts(
  cust_id INT,
  cust_name VARCHAR(255),
  account_no VARCHAR(255),
  account_balance INT
);

INSERT INTO cust_accounts(cust_id, cust_name, account_no, account_balance)
VALUES (2, 'Venzi', '58756500', 89876);

-- STEP:02 TRIGGER

CREATE or replace TRIGGER before_update_cust_accounts BEFORE
UPDATE ON
cust_accounts
FOR EACH ROW declare
    BEGIN
        IF:NEW.account_balance < 0 THEN:NEW.account_balance:= 0;
    END IF;
End;
```

- **Test Trigger:**

1. `SELECT * FROM cust_accounts`

CUST_ID	CUST_NAME	ACCOUNT_NO	ACCOUNT_BALANCE
2	Venzi	58756500	89876

2. `UPDATE cust_accounts SET account_balance=-90 WHERE cust_id=2;`
3. SELECT * FROM cust_accounts WHERE cust_id = 2;

CUST_ID	CUST_NAME	ACCOUNT_NO	ACCOUNT_BALANCE
2	Venzi	58756500	0

Before Delete Trigger: A BEFORE DELETE Trigger means that Oracle will fire this trigger before the DELETE operation is executed.

Create tables given below. Write a trigger that makes detail entry of entry made in salaries table.

```
-- STEP:01 TABLE

CREATE TABLE Salaries (
  employeeNumber INT PRIMARY KEY,
```

```
  validFrom DATE NOT NULL,
  amount DEC(12, 2) NOT NULL
);

CREATE TABLE SalaryArchives (
  employeeNumberINT PRIMARY KEY,
  validFrom DATE NOT NULL,
  amount DEC(12, 2) NOT NULL,
  deletedAt TIMESTAMP
);

-- STEP:02 TRIGGER

CREATE orreplace TRIGGER before_salaries_delete
BEFORE DELETE
ON salaries FOR EACH ROW
    DECLARE
    BEGIN
        INSERT INTO SalaryArchives(employeeNumber,validFrom,amount,deletedat)
        VALUES(:OLD.employeeNumber,:OLD.validFrom,:OLD.amount,(select current_date f
        dbms_output.put_line('Before Trigger Successfully executed');
END;
```

- **Test Trigger:**

1. `SELECT * FROM Salaries`

EMPLOYEENUMBER	VALIDFROM	AMOUNT
1	14-MAR-22	8977
2	15-MAR-22	6977

2. `DELETE FROM Salaries WHERE employeeNumber = 1;`
   "Before Trigger Successfully Executed. 1 row(s) deleted."
3. `SELECT * FROM Salaries`

EMPLOYEENUMBER	VALIDFROM	AMOUNT
2	15-MAR-22	6977

4. As soon as you delete data from salaries table entry is made in SalaryArchives
   `SELECT * FROM SalaryArchives`

EMPLOYEENUMBER	VALIDFROM	AMOUNT	DELETEDAT
1	14-MAR-22	8977	17-MAR-23
2	15-MAR-22	6977	17-MAR-23

Statement Level Trigger: A statement-level trigger is fired whenever a trigger event occurs on a table regardless of how many rows are affected.

- In other words, a statement-level trigger executes once for each transaction.
- For example, if you update 1000 rows in a table, then a statement-level trigger on that table would only be executed once.

- By default, the statement `CREATE TRIGGER` creates a statement-level trigger when you omit the `FOR EACH ROW` clause.

  Create table given below and make a statement level trigger.

```
-- STEP:01 TABLE
create table customersCredit (
  cid number(5) primary key,
  name varchar(20),
  address varchar(20),
  website varchar(20),
  creditlimit number(5)
);

insert into customersCredit
values (1, 'Manika', 'Surat', 'www.amazon.com', 50000);

-- STEP:02 TRIGGER

CREATE OR REPLACE TRIGGER customers_credit_trg BEFORE
UPDATE OF creditlimit
ON customersCredit
    DECLARE
        l_day_of_month NUMBER;
    BEGIN
        l_day_of_month := EXTRACT(DAY FROM sysdate);
    END;
```

- **Test Trigger:**

1. If you update table and current date is not between 28 to 31 then table will be updated.: `UPDATE customersCredit SET creditlimit = 9000 WHERE cid = 1`
   "1 row(s) updated"
2. If you update table and current date is between 28 to 31 then table will not be updated and application error is raised as in code: `UPDATE customersCredit SET creditlimit = 9000 WHERE cid = 1`

```
ORA-20100: Cannot update customer credit from 28  to 31
ORA-06512: at "SYSTEM.CUSTOMERS_CREDIT_TRG", line 8
ORA-04088: error during execution of trigger 'SYSTEM.CUSTOMERS_CREDIT_TRG'
```

Questions

1. What are triggers?
2. Explain Types of Triggers with syntax.
3. Explain the advantages and the needs of triggers.
4. Explain types of Triggers.
5. Explain statement level triggers with example.
6. Explain features of triggers and how can we access old and new values using trigger?

Join Us on Telegram @SOU BCA