CHAPTER -15
TRIGGERS

What is a Trigger :

A trigger is a stored procedure that defines an action the database automatically initiates when some database related events such as INSERT, UPDATE OR DELETE occurs.

Why Triggers:

A trigger is a fragment of code that you tell Oracle to run before or after a table is modified. A trigger has the power to :-

■*make sure that a column is filled in with default information*

■*make sure that an audit row is inserted into another table*

■*after finding that the new information is inconsistent with other stuff in the database, raise an error that will cause the entire transaction to be rolled back*

■The different types of integrity constraints provide a *declarative* mechanism to associate "simple" conditions with a table such as a primary key, foreign keys or domain constraints.

■Complex integrity constraints that refer to several tables and attributes cannot be specified within table definitions. *Triggers*,in contrast, provide a procedural technique to specify and maintain integrity constraints.

Triggers Vs Procedures:

■Triggers Execute Implicitly while Procedure execute explicitly.

■It do not accept arguments while Procedures may or may not have arguments.

■Triggers are fired for DML(Insert, Update or Delete) statements while procedures execute all DML statements including SELECT.

Trigger Vs constraints:

■Triggers affect only the row added after the trigger is enabled.

■Affects all the rows in a table

Syntax

CREATE [OR REPLACE] TRIGGER *trigger_name*
Before/ after insert/update/ delete
[of columnname]
ON *table_name*
[For each Row]
BEGIN
SQL statements;
END [triggername];

The structure of a row-level

```
CREATE OR REPLACE TRIGGER ***trigger name***
  ***when***
  ON
  ***which table***
FOR EACH ROW
  ***conditions for firing***
begin
  ***stuff to do***
end;
```

Components of a trigger definition :

trigger name

create [**or replace**] **trigger** <trigger name>

- *trigger time point*
  before | **after**
- *triggering event(s)*
  insert or update [**of** <column(s)>] **or delete on** <table>
- *trigger type* (optional)
  for each row
- *trigger restriction* (only for **for each row** triggers !)
  when (<condition>)
- *trigger body*
  <PL/SQL block>

Executing Triggers

When using SQL*Plus, you have to provide a / character to get the program to evaluate a trigger or PL/SQL function definition. You then have to say **"show errors"** if you want SQL*Plus to print out what went wrong. Unless you expect to write perfect code all the time, it can be convenient to leave these SQL*Plus incantations in your .sql files.

Types of Triggers

- Row level triggers
- Statement Level Trigger
- Before and after Trigger
- Instead of Trigger
- Trigger on system events and user events

Example: 1

```
SQL> Create or replace trigger  empcount
       After insert  on emp
      For each row
      Declare
            n integer;
     Begin
         Select count(*) into n from emp;
               dbms_output.put_line('total no. of records in a table is : '||n);
End;
```

Accessing coloumn values :

- :old .<colomn name>
- :new. <Colomn name>

* * Points to Remember * *

* Only with a row trigger it is possible to access the attribute values of a tuple before and after the modification (because the trigger is executed once for each row).

* For an **update** trigger, the old attribute value can be accessed using **:old**.<column> and the new attribute value

can be accessed using **:new**.<column>.

* For an **insert** trigger, only **:new**.<column> can be

Used.

* for a **delete** trigger only **:old**.<column> can be used (because there exists no old, respectively, new value of the tuple). In these cases, **:new**.<column> refers to the attribute value of <column> of the inserted tuple, and **:old**.<column> refers to the attribute value of

<column> of the deleted tuple.

In a row trigger thus it is possible to specify comparisons between old and new attribute values in the PL/SQL block,

e.g., "**if :old**.SAL < **:new**.SAL **then** . . . ".

If for a row trigger the trigger time point **before** is specified, it is even possible to modify the new values of the row, e.g., **:new**.SAL := **:new**.SAL * 1.05 or **:new**.SAL := **:old**.SAL.

*Such modifications are not possible with **after** row triggers.*

Example : 1

```
SQL> Create or replace trigger  empcount
        Before delete  on emp
       For each row
       Declare
            n integer;
      Begin
          select count(*) into n from emp;
             Dbms_output.put_line('total no. of   records in a table is : '||n);
End;
```

Example : 2

```
SQL> Create or replace trigger  EMPUPD
        Before update on emp
       For each row
Begin
       if :new.salary<:old.salary then
     Dbms_output.put_line('Salary can not be reduced');
End;
```

Example : 3

statement level trigger-

```
SQL> Create or replace trigger  EMPUPD
        Before update on emp
Begin
        if :new.salary<:old.salary then
      Dbms_output.put_line('Salary can not be reduced');
End;
```

Example 4:

```
SQL>    Create or replace trigger  EMPUPD
        After update on emp
        n number;
        Begin
        select count(*)  into n from emp;
        Dbms_output.put_line('Total Records in table EMP :'||n);
        End;
```

- **Enabling a Trigger is:**

ALTER TRIGGER *trigger_name* ENABLE;

For example:
If you had a trigger called orders_before_insert,
you could enable it with the following command:
ALTER TRIGGER orders_before_insert ENABLE;

Disable a Trigger
syntax :

ALTER TRIGGER *trigger_name* DISABLE;
For example:
ALTER TRIGGER orders_before_insert DISABLE;

Drop a Trigger
syntax :
DROP TRIGGER *trigger_name*;

For example:
DROP TRIGGER orders_before_insert;

Example:

```
create or replace trigger check_budget_EMP
after insert or update of SAL, DEPTNO on EMP
declare
cursor DEPT_CUR is select DEPTNO, BUDGET from DEPT;
 DNO DEPT.DEPTNO%TYPE;
ALLSAL DEPT.BUDGET%TYPE;
DEPT_SAL number;
begin
open DEPT_CUR;
```

loop
fetch DEPT_CUR **into** DNO, ALLSAL;
exit when DEPT_CUR**%NOTFOUND**;
select sum(SAL) **into** DEPT_SAL **from** EMP **where** DEPTNO = DNO;
if DEPT_SAL > ALLSAL **then**
raise_application_error(-20325, 'Total of salaries in the department '|| to_char(DNO) || ' exceeds budget');
end if;
end loop;
close DEPT_CUR;
end; /

More about triggers :

Triggers are not exclusively used for integrity maintenance. They can also be used for

• Monitoring purposes, such as the monitoring of user accesses and modifications on certain sensitive tables.

• Logging actions, e.g., on tables:

Contd..

create trigger LOG EMP
after insert or update or delete on EMP
begin
if inserting then
insert into EMP LOG **values**(**user**, 'INSERT', **sysdate**);
end if ;
if updating then
insert into EMP LOG **values**(**user**, 'UPDATE', **sysdate**);
end if ;
if deleting then
insert into EMP LOG **values**(**user**, 'DELETE', **sysdate**);
end if ;
end;

By using a row trigger, even the attribute values of the modified tuples can be stored in the table EMP LOG.

• automatic propagation of modifications. For example, if a manager is transferred to another department, a trigger can be defined that automatically transfers the manager's employees to the new department.

More about Triggers

If a trigger is specified within the SQL*Plus shell, the definition must end with a point "." in the last line. Issuing the command **run** causes SQL*Plus to compile this trigger definition.

A trigger definition can be loaded from a file using the command @. Note that the last line in the file must consist of a slash "/".

A trigger definition cannot be changed, it can only be re-created using the **or replace** clause.

The command **drop** <trigger name> deletes a trigger.

After a trigger definition has been successfully compiled, the trigger automatically is enabled.

The command **alter trigger** <trigger name> **disable** is used to deactivate a trigger. All triggers defined on a table can be (de)activated using the command

alter table <Tablename> **enable** | **disable all trigger**;

The data dictionary stores information about triggers in the table USER TRIGGERS. The information includes the trigger name, type, table, and the code for the PL/SQL block.

Difference b/w For and Do Loops: ***When No. of repetitions are known then For loop is used, and if the No. of iterations are unknown then do loops are used.***

Difference b/w While and Until: ***While means as long as the condition is true, the loop execute the body***
Whereas Until means as long as the condition is not true, the loop repeats

Exiting from Loop: ***Exit statement helps to terminate any of the loops directly.***
EXIT DO : To terminate any Do loop
EXIT FOR : To terminate for loop

Use of For Each … Next Loop: It is used to repeat a group of statements for each element in a dynamic array as we are not sure about the size of the array.

Two Basic Operations on Arrays: ***Traversing means processing each element of the array***
Searching means to find a given element in array.

Calling Procedure : *It's a procedure that calls another procedure.*
Called/Caller Procedure : *The procedure being called is known as Called / Caller Procedure.*

Actual Parameters : *The parameters provided by calling procedures are actual.*
Formal Parameters : *The parameters received by called procedures are formal.*

A sub procedure may call in two ways:

With a call statement -	*Call procedure-name (actual arguments list)*	*Eg: Call abc (x , y)*
Without call statement-	*procedure – name actual arguments*	*Eg: abc x, y*

If Private/Public keyword is not specified with a procedure then the procedure becomes ***Public****.*
The value being returned by the function is assigned to the function name, which automatically returns it to the calling procedure or function. A function may return only one value.

**Sub procedure does not return a value, so a call to a sub procedure is a complete statement.*
** Function procedure returns a value, so a call to a function procedure is part of an expression.*

In a procedure, optional parameters are declared in argument list from right hand side.

Sub OptProcedure(ByVal X as Integer, ByVal Y as Integer**, ByVal Optional Z as Integer)**

VB Passes an argument ***by Reference by default.***

Exit sub *and* ***Exit Function*** *statements can be used to Exit from a sub procedure or a function procedure.*

If a variable is declared as ***PUBLIC A as Integer*** *in* ***form1*** *and it's* ***value is 20****, then it can be used in form2 as* ***form1.A****.*

List the variable scopes in decreasing lifespan: **PUBLIC, MODULE, STATIC, LOCAL**

Try this:

```
Sub MyProc1 ( )
        Dim A as Integer
        A = 12
        Print A
        Call MyProc2 ( A )
        A = A + 2
        Print A
End Sub
Sub MyProc2 ( B as Integer)
        Print B
        B = B + 10
        Print B
End Sub
```

O/P is:

12
12
22
24

When a number is converted to a string, a leading space is always reserved for its sign.

```
St = Str (198)          ' Gives "  198"
St = Str (-198)         ' Gives "-198"
```

Cint ( ) function returns truly rounded number. Eg. : Print CInt (-14.8) will print -15.

* * *