Advance SQL

Part-1

query := 'INSERT INTO dept_new VALUES (:dept_no, :dept_name, :loc)';

This example inserts a new row for which the column values are in the PL/SQL variables dept_no, dept_name, and location. Table shows sample code that accomplishes this DML operation using the DBMS_SQL package and native dynamic SQL.

Table DML Operation Using the DBMS_SQL Package and Native Dynamic SQL

1. **Code of DBMS_SQL DML Operation**

```
SQL> DECLARE
  2    stmt_str VARCHAR2(350);  my_cur NUMBER;
  3    deptid NUMBER := 101;  deptname VARCHAR2(20);
  4    location VARCHAR2(20);myresources VARCHAR2(20);
       rows_processed NUMBER;
  5  BEGIN
  6    stmt_str := 'INSERT INTO departments VALUES(:did,
:dname,

          :location,:resources)';
  7    my_cur := DBMS_SQL.OPEN_CURSOR;
8DBMS_SQL.PARSE(my_cur, stmt_str,
DBMS_SQL.NATIVE);
  9      -- supply binds
 10    DBMS_SQL.BIND_VARIABLE     (my_cur, ':did',
deptid);
 11    DBMS_SQL.BIND_VARIABLE     (my_cur, ':dname',
deptname);
 12    DBMS_SQL.BIND_VARIABLE     (my_cur, ':location',
location);
 13    DBMS_SQL.BIND_VARIABLE     (my_cur, ':resources',
myresources);
 14
 15rows_processed := dbms_sql.execute(my_cur);
 16      -- execute
 17    DBMS_SQL.CLOSE_CURSOR(my_cur); -- close
 18END;
 19 /
```

2. **Code of Native Dynamic SQL DML Operation:**

```
SQL> DECLARE
  2    stmt_str VARCHAR2(350);   deptid NUMBER := 102;
  3    deptname VARCHAR2(20);   location VARCHAR2(20);
4     myresource  VARCHAR2(20);

  5  BEGIN
```

```
 6    stmt_str := 'INSERT INTO departments VALUES
 7         (:did, :dname, :location,:resources)';
8EXECUTE IMMEDIATE stmt_str
 9      USING deptid, deptname, location ,myresource;
10END;
11  /
```

14.13 USE OF DYNAMIC SQL DIFFERENT LANGUAGES:

The dynamic SQL is also supported in various database languages with their language specifications. We can call dynamic SQL from other languages as:

- If we use C/C++, we can call dynamic SQL with the Oracle Call Interface (OCI), or we can use the Pro*C/C++ pre-compiler to add dynamic SQL extensions to our C code.
- If we use COBOL, we can use the Pro*COBOL pre-compiler to add dynamic SQL extensions to our COBOL code.
- If we use Java, we can develop applications that use dynamic SQL with JDBC.

If we have an application that uses OCI, Pro*C/C++, or Pro*COBOL to execute dynamic SQL, we should consider switching to native dynamic SQL inside PL/SQL stored procedures and functions. The network round-trips required to perform dynamic SQL operations from client-side applications might decrease performance. Stored procedures can reside on the server, eliminating the network overhead. We can call the PL/SQL stored procedures and stored functions from the OCI, Pro*C/C++, or Pro*COBOL application.

14.14 QUESTIONS

1. State the execution flow of SQL in PL/SQL Subprograms.
2. How to execute PL/SQL Block Dynamically?
3. Write short note on Dynamic SQL.
4. How to execute Dynamic queries?
5. What is Native Dynamic SQL?
6. Write short note on DBMS_SQL Package.
7. State the Advantages of Native Dynamic SQL.
8. State the Advantages of DBMS_SQL Package.
9. Where we can use Dynamic SQL other than PLSQL?

Practice Questions:

10. Write a Simple example for DML Operation Using Native Dynamic SQL.
11. Write a Simple example for DDL Operation Using Native Dynamic SQL.
12. Write a Simple example for Multiple-Row Query Using Native Dynamic SQL.
13. Use the DBMS_SQL package in the above examples.

14.15 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

15

TRIGGERS

Unit Structure

15.1 OBJECTIVES

After completing this chapter, you will be able to:

- Understand the Fundamentals of Triggers
- Create and use Triggers

- Understand the types of Triggers
- Understand the Execution Hierarchy of PL/SQL Trigger
- Creating DML and DDL Triggers
- View , Alter and Drop the Triggers
- Understand the Cyclic Cascading in a Triggers
- Understand the Privileges Required to Use Triggers

2. DEFINING A TRIGGER:

The triggers play an important role while validating the SQL and PLSQL queries on automatic basis depending on the particular condition. It executes or fired like a definite event every time. A trigger can be defined as automatic code execution on a particular event of a database. A trigger is a PL/SQL block structure or a subprogram which is fired or executed when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically at predefined timing and event, when an associated DML statement is executed. Triggers are physically stored in database.

A trigger stored in the database can include SQL and PL/SQL or Java statements to run as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger. Triggers are implicitly fired by Oracle when a triggering event occurs, no matter which user is connected or which application is being used. Triggers are useful in achieving security and auditing. It also maintains data integrity and referential integrity.

3. INSIDE THE TRIGGERS

The triggers give the dynamic approach to our SQL script. Before we create and use the trigger, the user SYS must run a SQL script commonly called DBMSSTDX.SQL. The proper name and location of this script depend on our operating system. Before starting with the triggers we must consider following points.

- We must have the CREATE TRIGGER system privilege, to create a trigger in our own schema on a table.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*schema*. SCHEMA), we must have the CREATE ANY TRIGGER system privilege.
- To create a trigger on DATABASE, we must have the ADMINISTER DATABASE TRIGGER system privilege.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.
In short we must have maximum privileges to deal with the triggers.

15.4 THE DATABASE TRIGGERS & APPLICATION TRIGGERS:

There are various types of triggers. Triggers can be categorized as Database triggers and Application triggers on the basis of scope of usage of triggers. The Database triggers are activated on any event occurring in the database, while application trigger are restricted to an application. Database triggers can be created on top of table, view, schema or database. Timings for table or view can be before and after a DML operation, while those on schema and database
can be logging in and log off.

A. Triggers with DML:

The following are the three criteria which must be kept in mind before creating and using the DML trigger.

1. There can be three possible DML actions on data i.e. INSERT, UPDATE or DELETE. These are events for the DML triggers.
2. A simultaneous action can be performed either before or after an event. This serves as timing for DML triggers.
3. Whether the trigger action must be at DML statement level or at affected row level, decides the level of a trigger.

   After the timing, event and level are set, trigger body must be created to implement the triggering logic.

Important Note: *The size of a trigger cannot be greater than 32 KB.*

Syntax of Triggers:
The Syntax for creating a trigger is:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF column_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
```

```
[FOR EACH ROW]
WHEN (condition)
BEGIN
--- The SQL Code // application logic goes here
END;
/
```

Understanding the Syntax:

- ***CREATE [OR REPLACE] TRIGGER trigger_name*** :- This line creates a trigger with the given name or overwrites an existing trigger with the same name. It is a compulsory part of syntax.
- ***{BEFORE | AFTER | INSTEAD OF }*** - This line indicates at what time the trigger should 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}*** - This line determines the triggering event. 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]*** - This statement is used with update triggers. This statement is used when we want to trigger an event only when a specific column is updated.
- ***[ON table_name]*** - This statement identifies the name of the table or view to which the trigger is associated.
- ***[REFERENCING OLD AS o NEW AS n]*** - This statement is used to reference the old and new values of the data being changed. By default, we 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. We cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- ***[FOR EACH ROW]*** - This statement is 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)*** - This statement is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: If we want to avoid the duplicate entry of the field other than primary field then we can create trigger to check that particular entry. 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 'price_trace' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'price_trace ' table

```
SQL> CREATE TABLE price_trace
16    (product_id number(5), product_name varchar2(32),
17     supplier_name varchar2(32),
4  unit_price number(7,2) );

Table created.

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

Table created.

SQL> insert into product values(1312,'Wooden_Door','Galaxy',950);

1 row created.

SQL> insert into product values(1313,'Plastic_Door','Tanmay',1950);

1 row created.

SQL> insert into product values(1314,'Metal_Door','Sun',11450);

1 row created.
```

2)Create the my_price_trace trigger and execute it. SQL>

```
CREATE or REPLACE TRIGGER  my_price_trace
  2BEFORE UPDATE OF unit_price
  3  ON product
  4  FOR EACH ROW
  5  BEGIN
  6  INSERT INTO price_trace
  7               VALUES       (:old.product_id,       :old.product_name,
:old.supplier_name, :old.unit_price);
  8END;
  9  /

Trigger created.
```

3) Lets update the price of a product.

```
          SQL>  UPDATE  product  SET  unit_price  =  900  WHERE
product_id = 1312;
          1 row updated.
```

Once the above update query is executed, the trigger fires and updates the 'price_trace' table. We can view the result using following statements.

SQL> select * from price_trace;

PRODUCT_ID	PRODUCT_NAME	SUPPLIER_NAME	UNIT_PRICE
1312	Wooden_Door	Galaxy	950

4) If we **ROLLBACK** the transaction before committing to the database, the data inserted to the table is also rolled back.

15.5 CLASSIFICATION OF PL/SQL TRIGGERS:

There are two types of triggers based on the level it is triggered.

1)Trigger on Row level: - The Row level trigger fires automatically when any other query makes any change in the any single table row. The row level trigger is fired each time the table is affected by the triggering statement. For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not executed at all. Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected.

2)Trigger on Statement level: - The statement trigger is fired once on behalf of the triggering Statement, regardless of the number of rows in the table that the triggering statement affects (even if no rows are affected). For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.

Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected. For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

15.6 DIFFERENCE BETWEEN BEFORE & AFTER TRIGGERS:

The Before and After triggers are related with the timing of firing them or we can say that when to execute them. When defining a trigger, we can specify the trigger timing. Means, we can

specify whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers.

1)BEFORE Triggers: The BEFORE triggers executes the trigger action before the triggering statement. The BEFORE triggers are used when the trigger action should determine whether the triggering statement should be allowed to complete. By using a BEFORE trigger for this purpose, we can eliminate unnecessary processing of the triggering statement and its ultimate rollback in cases where an exception is raised in the trigger action. BEFORE triggers are also used to derive specific column values before completing a triggering INSERT or UPDATE statement.

2)AFTER Triggers: The AFTER triggers execute the trigger action after the triggering statement is executed. The AFTER triggers are used when we want the triggering statement to complete before executing the trigger action. If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

We can have multiple triggers of the same type for the same statement for any given table. For example we may have two *BEFORE STATEMENT* triggers for *UPDATE* statements on the *EMPLOYEE* table. Multiple triggers of the same type permit modular installation of applications that have triggers on the same tables. We can design our own *AFTER ROW* trigger in addition to the Oracle-defined *AFTER ROW* trigger.

We can create as many triggers of the preceding different types as we need for each type of DML statement (INSERT, UPDATE, or DELETE). For example, suppose we have a table Payment, and we want to know when the table is being accessed and the types of queries being issued.

15.7 EXECUTION SEQUENCE OF PL/SQL TRIGGER:

There are some rules of execution of triggers. The following sequence is followed when a trigger is fired.
1)The BEFORE statement trigger executes / 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. These 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_chk' which we can use to store messages when triggers are fired.

```
SQL> CREATE TABLE product_chk (Message varchar2(50),
Current_Date date  );
Table created.
```

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_chk' before a SQL update statement is executed, at the statement level.

```
SQL>  CREATE  or  REPLACE  TRIGGER
Before_Update_product
  2BEFORE
  3  UPDATE ON product
  4  Begin
  5  INSERT INTO product_chk
  6  Values('Before update, statement level trigger', sysdate);
  7END;
  8  /

Trigger created.
```

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

```
SQL>  CREATE  or  REPLACE  TRIGGER
Before_Upddate_Row_product
  2   BEFORE
  3   UPDATE ON product
  4   FOR EACH ROW
  5   BEGIN
  6   INSERT INTO product_chk
  7   Values('Before update row level trigger',sysdate);
  8   END;
  9   /

Trigger created.
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_chk' after a SQL update statement is executed, at the statement level.

```
SQL>  CREATE or REPLACE TRIGGER
After_Update_product
  2   AFTER
```

```
3  UPDATE ON product
4  BEGIN
5  INSERT INTO product_chk
6  Values('After update, statement level trigger', sysdate);
7  End;
8  /
```

Trigger created.

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

```
SQL> CREATE or REPLACE TRIGGER
After_Update_Row_product
 2  AFTER
 3  insert On product
 4  FOR EACH ROW
 5  BEGIN
 6  INSERT INTO product_chk
 7  Values('After update, Row level trigger',sysdate);
 8  END;
 9  /
```

Trigger created.

Now let's execute a update statement on table item.

```
SQL> UPDATE product SET unit_price = 850
 2  WHERE product_id in (1312,1314);
```

2 rows updated.

We can check the data in 'product_chk' table to see the order in which the trigger is fired.

```
SQL>  SELECT * FROM product_chk;
```

Output:

MESSAGE	CURRENT_DATE
Before update, statement level	29-AUG-12
Before update row level trigger	29- AUG -12
Before update row level trigger	29- AUG -12
After update, statement level triggers	29- AUG -12

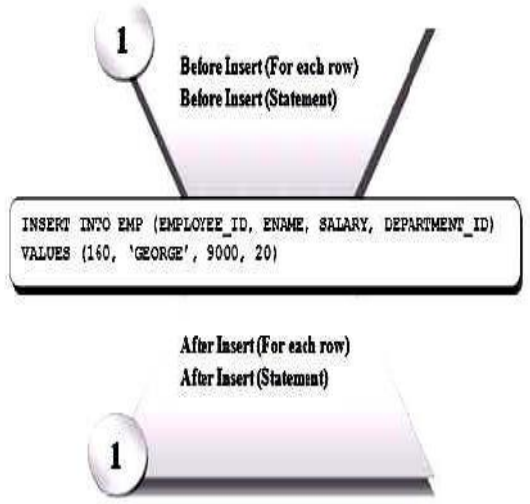
The above result shows 'before update' and 'after update' row level events have occurred twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per SQL statement.

The above rules apply similarly for INSERT and DELETE statements.

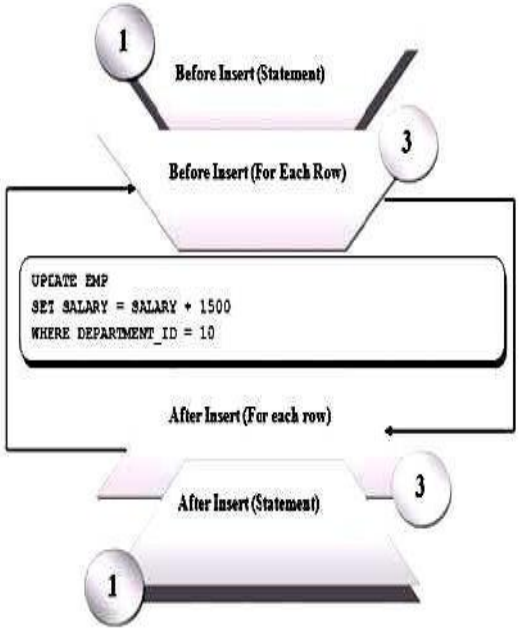
15.8 DIFFERENCE BETWEEN STATEMENT LEVEL AND ROW LEVEL TRIGGERS :

A)Statement-Level Triggers: These are the default triggers. These are fired only once when the triggering event occurs. These triggers does not have any effect that any row affected or not by the update or insert statement.

B)Row-Level Triggers: To fire these types of triggers FOR EACH ROW specification is required. These triggers are fired only when the rows affected by an event. If no rows are affected, it will not fire.



Trigger-Firing Sequence: Single-Row Manipulation



Trigger-Firing Sequence: Multi-Row Manipulation

Creating a DML Statement Trigger

we can create a statement level trigger as below. The trigger fires before the INSERT action on EMPLOYEE table. As per the trigger action, it inserts a record in Employee Log table, with current date, action and remarks.

```
SQL> create table employee(empid int,empname char(20),
  2          empsal number(7,2), jobtitle char(100));

Table created.
```

```
SQL> create table trace_emp(empid int,status char(2),
  2          actdate date,act char(20),remark char(100));

Table created.

SQL> CREATE OR REPLACE TRIGGER trace_employee
2BEFORE INSERT ON employee
  3  BEGIN
  4  INSERT INTO  trace_emp(empid, status, actdate, act, remark)
5VALUES(2, 'P', SYSDATE, 'CREATE', 'you are with the
triggers');
  6END;
  7  /

Trigger created.

SQL> INSERT INTO EMPLOYEE (empid, empname, empsal,
jobtitle)
  2  VALUES(4, 'SONALI', 34500, 'manager');

1 row created.


SQL> SELECT * FROM trace_emp;

    EMPID      ST       ACTDATE         ACT           REMARK
 ----------  ------  -----------------  ----------  ----------------------------------------
       2          P        29-AUG-12      CREATE     you are with the triggers
```

Conditional Predicates to Detect the Trigger DML operation:

We can use conditional predicate with the DML trigger operations. For a specific timing, if all the events have to be tested instead of creating three different DML triggers, oracle provides DML predicates to be used in trigger body. The available predicates can be INSERTING, UPDATING or DELETING. For example, below trigger body shows the usage of DML predicates. Note the event specification and handling.

```
CREATE OR REPLACE TRIGGER my_trigg_trace_ emp
BEFORE INSERT OR UPDATE OR DELETE ON employee
BEGIN

IF INSERTING THEN
…
ELSIF UPDATING THEN
…
ELSIF DELETING
…  END
IF;
```

```
END;
/
```

Output>>TRIGGER created.

15.9 BUILDING A DML ROW LEVEL TRIGGER

The DML row level trigger EMPLOYEE_MEMBERSHIP deletes the membership record for every employee record which gets deleted.

```
SQL> CREATE OR REPLACE TRIGGER EMPLOYEE_MEMBERSHIP
  2BEFORE DELETE ON EMPLOYEE
  3  BEGIN
  4  FOR EACH ROW
  5   DELETE FROM EMP_MEMBERSHIP
  6   WHERE EMPID=:OLD.EMPID;
  7END;
  8   /
```

Output>>TRIGGER created.

10. DDL TRIGGER CREATION:

Following example creates an AFTER statement trigger on any DDL statement CREATE. Such a trigger can be used to audit the creation of new data dictionary objects in our schema.

```
CREATE TRIGGER audit_db_field AFTER CREATE
  ON SCHEMA pl/sql_block
```

11. CALLING A PROCEDURE IN A TRIGGER BODY:

In following example we could create the check_salary trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume we have defined a procedure check_salary in the hr schema, which verifies that an employee's salary is in an appropriate range. Then we could create the trigger check_salary as follows:

```
CREATE TRIGGER check_salary
BEFORE INSERT OR UPDATE OF empsal, jobtitle ON employees
   FOR EACH ROW
   WHEN (new. jobtitle <> 'DEVELOPER')
   CALL check_salary(:new.jobtitle, :new.empsal, :new.empname)
```

The procedure check_salary could be implemented in PL/SQL, C, or Java. Also, we can specify: OLD values in the CALL clause instead of: NEW values.

12. BUILDING A DATABASE EVENT TRIGGER:

Following example demonstrates the basic syntax for a trigger to log all errors. The hypothetical PL/SQL block does some special processing for a particular error (invalid logon, error number 1017). This trigger is an AFTER statement trigger, so it is fired after an unsuccessful statement execution, such as unsuccessful logon.

```
CREATE TRIGGER errlog AFTER SERVERERROR ON
DATABASE
  BEGIN
    IF (IS_SERVERERROR (1017)) THEN
      <special processing of logon error>
    ELSE
      <log error number>
    END IF;
  END;
/
```

13. CREATION OF A SCHEMA TRIGGER:

Following example creates a BEFORE statement trigger on the sample schema hr. When a user connected as hr attempts to drop a database object, the database fires the trigger before dropping the object.

```
CREATE OR REPLACE TRIGGER drop_resist_trigger
  BEFORE DROP ON hr.SCHEMA
  BEGIN
    RAISE_APPLICATION_ERROR ( num => -20000,   msg =>
'not able to drop object');
  END;
/
```

14. IDENTIFIERS (OLD AND NEW)

The OLD and NEW are identifiers which carry a record value before and after the DML event. The record values can be referred by prefixing a column value with the corresponding identifier. Below table shows the OLD and NEW values within each triggering event.

Event	OLD value	NEW value
INSERT	NULL	Current value

UPDATE	Old value of record	New value of record
DELETE	Record value before delete operation	NULL

Example:
The trigger below archives an employee record if salary is incremented by more than 2000. Note that the increment is checked by the WHEN clause condition.

```
SQL> CREATE OR REPLACE sal_incr
2BEFORE UPDATE OF empsal ON employee
  3  FOR EACH ROW
  4  WHEN(OLD.empsal  -  NEW.empsal > 2000)
  5  BEGIN
6INSERT INTO EMP_ARCHIVE (id, empid, OLD_SAL,
NEW_SAL, REVISED_DT)
      VALUES
  7  (SQ_ARC.NEXTVAL, :OLD.EMPID,:OLD.SALARY,
:NEW.SALARY, SYSDATE);
  8 END;
  9 /
```


15.15 INSTEAD OF TRIGGERS (VIEW TRIGGERS):

While database programming sometimes there may be situation that the triggers has two options and it must fired with alternate options. The INSTEAD OF trigger satisfy the condition. Triggers on views are known as INSTEAD OF triggers. They are known by their name because they skip the current triggering event action and perform alternate one. Other reason could be that only timing mode available in such triggers is INSTEAD OF. It is required for the complex view because it is based on more than one table. Any DML on complex view would be successful only if all key columns, not null columns are selected in the view definition. Alternatively, INSTEAD OF trigger can be created to synchronize the effect of DML across all the tables.

Instead of trigger is a row level trigger and can be used only with a view, and not with tables. For

Example, following view ORD_VU is created on top of ORDERS and WAREHOUSE tables. If
RET_LIMIT is attempted for update in the view, a record must be added to WAREHOUSE_HISTORY table and new value must be updated in the WAREHOUSE table.

```
SELECT O.ID, O.QTY, O.ORD_DATE, P.SITE_ID,
P.RET_LIMIT
FROM ORDERS O, WAREHOUSE P
WHERE O.SITE_ID=P.SITE_ID
CREATE OR REPLACE TRIGGER T_UPD_ORDERVIEW
INSTEAD OF UPDATE ON ORD_VU
BEGIN
  INSERT INTO WAREHOUSE_HISTORY
  (SITE_ID, OLD_RET_LIMIT, NEW_RET_LIMIT,
UPD_DATE, UPD_USER)
  VALUES
   (:OLD.SITE_ID, :OLD_RET_LIMIT, :NEW.RET_LIMIT,
SYSDATE, USER);
   UPDATE WAREHOUSE
  SET RET_LIMIT = :NEW.RET_LIMIT
WHERE SITE_ID = :OLD.SITE_ID;
END;
/
```

16. LISTING OF TRIGGERS INFORMATION:

We can see all our user defined triggers by doing a select statement on USER_TRIGGERS. This will gives us clear idea about the trigger and its structure.
For example:

```
SELECT TRIGGER_NAME FROM USER_TRIGGERS;
```

Above statement produces the names of all triggers. We can also select more columns to get more detailed trigger information. We can do that at our own relaxation, and explore it on our own.

17. ALTERING A TRIGGER:

There is facility to change the trigger code and conditions. If a trigger seems to be getting in the way, and we don't want to drop it, just disable it for a little while, we can alter it to disable it. Note that this is not the same as dropping a trigger; after we drop a trigger, it is gone.

The general format of an alter would be something like this:

```
ALTER TRIGGER trigger_name [ENABLE|DISABLE];
```

For example, let's say that with all our troubles, we still need to modify the DOB of 'SONALI SAMBARE '. We cannot do this since we have a trigger on that table that prevents just that. So, we can disable it...

```
ALTER TRIGGER PERSON_DOB DISABLE;
```

Now, we can go ahead and modify the DOB :-)

```
UPDATE PERSON SET DOB = SYSDATE WHERE NAME = 'YASHASHREE SAMBARE';
```

We can then re-ENABLE the trigger.

```
ALTER TRIGGER PERSON_DOB ENABLE;
```

If we then try to do the same type of modification, the trigger kicks and prevents us from modifying the DOB.

- **Syntax for removing Triggers:**

For removing the trigger we have to use following syntax.

```
DROP TRIGGER trigger_name;
```

15.18 KNOWING 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  DESCRIPTION                     VARCHAR2(8)
ACTION_TYPE                             VARCHAR2(4000)
TRIGGER_BODY                            VARCHAR2(11)
                                        LONG
```

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

```
SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_product';
```

The above SQL query provides the header and body of the trigger 'Before_Update_Stat_product'.
We can drop a trigger using the following command.

DROP TRIGGER trigger_name;

15.19 CYCLIC CASCADING in a TRIGGER:

Sometimes the triggers may create some critical situation. This is an undesirable situation where more than one trigger enters into an infinite loop. While creating a trigger we should ensure the situation does not exist.

Let's consider we have two tables 'inv' and 'product'. Two triggers are created.

1)The INSERT Trigger, triggerA on table 'inv' issues an UPDATE on table 'product'.

2)The UPDATE Trigger, triggerB on table 'product' issues an INSERT on table 'inv'.

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

15.20 BOUNDARIES ON TRIGGER CONDITIONS:

Trigger conditions are subject to the following restrictions:

1)If we specify this clause for a DML event trigger, then we must also specify FOR EACH ROW. Oracle Database evaluates this condition for each row affected by the triggering statement.

2)We cannot specify trigger conditions for INSTEAD OF trigger statements.

3)We can reference object columns or their attributes, or varray, nested table, or LOB columns. We cannot invoke PL/SQL functions or methods in the trigger condition.

15.21 TRIGGER EXCEPTIONS:

Triggers become part of the transaction of a statement, which implies that it causes (or raises) any exceptions, the whole statement is rolled back.

Think of an exception as a flag that is raised when an error occurs. Sometimes, an error or exception is raised for a valid reason. For example, to prevent some action that improperly modifies the database. Let's say that our database should not allow anyone to modify their DOB (after the person is in the database, their DOB is assumed to be static). Anyway, we can create a trigger that would prevent us from updating the DOB:

```
CREATE OR REPLACE
TRIGGER change_resist_id
BEFORE UPDATE OF empid ON employee
FOR EACH ROW
BEGIN
RAISE_APPLICATION_ERROR (-20000,'CANNOT
CHANGE DATE OF BIRTH');
END;
/
```

Notice the format of the trigger declaration. We explicitly specify that it will be called BEFORE UPDATE OF DOB ON PERSON. The next thing we should notice is the procedure call RAISE APPLICATION ERROR, which accepts an error code, and an explanation string. This effectively halts our trigger execution, and raises an error, preventing our DOB from being modified. An error (exception) in a trigger stops the code from updating the DOB. When we do the actual update for example

UPDATE PERSON SET DOB = SYSDATE;

We end up with an error, which says we CANNOT CHANGE DATE OF BIRTH.

UPDATE PERSON SET DOB = SYSDATE;

```
UPDATE PERSON SET DOB = SYSDATE
*
ERROR at line 1:
ORA-20000: CANNOT CHANGE DATE OF BIRTH
ORA-06512: at "PARTICLE.PERSON_DOB", line 2
ORA-04088: error during execution of trigger
'PARTICLE.PERSON_DOB'
```

We should also notice the error code of ORA-20000. This is our -20000 parameter to RAISE APPLICATION ERROR.

22. PRIVILEGES REQUIRED TO USE TRIGGERS:

To work with triggers we have to satisfy some conditions. To create a trigger in our schema:

- We must have the CREATE TRIGGER system privilege
- One of the following must be true:
  - We own the table specified in the triggering statement
  - We have the ALTER privilege for the table specified in the triggering statement
  - We have the ALTER ANY TABLE system privilege

To create a trigger in another schema, or to reference a table in another schema from a trigger in our schema:

- We must have the CREATE ANY TRIGGER system privilege.
- We must have the EXECUTE privilege on the referenced subprograms or packages.

To create a trigger on the database, we must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, we can drop the trigger but not alter it.

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger owner, not the privilege domain of the user issuing the triggering statement (this is similar to the privilege model for stored subprograms).

23. QUESTIONS:

1. Define Triggers. Explain the syntax of creating triggers in PL/SQL.
2. Explain the types of triggers.
3. Distinguish between BEFORE and AFTER Triggers.
4. Write the execution hierarchy of Triggers.
5. Distinguish between Statement Level and Row Level Triggers.
6. How to create DML statement Triggers.
7. Explain the conditional predicate to detect the Trigger DML operation.
8. Explain the creation of DML Row Level Trigger with example.
9. How to call procedure in Trigger body.
10. Explain the creation of Database Event Trigger with example.

11. Explain the creation of SCHEMA Trigger with example.
12. Explain Triggers on view.
13. How to View, Alter and Remove Triggers? Explain with examples.
14. Explain Cyclic Cascading in Triggers.
15. List and Explain the Restrictions on Trigger Conditions.
16. Write short note on Trigger Exceptions.
17. List and Explain Privileges Required to Use Triggers. Create a student view for student's personal information.
18. Create a view for Teacher and change it to select all teachers having subject I.T...

15.24 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

Thank You