Chapter 10

Lecture for Week 10,11

10.1 Triggers

Before getting into triggers it is important to discuss few points about one topic:*views*. For information about views in details See: Orale 11g Complete Reference,Chapter 17

Views: Views are known as logical tables. They represent the data of one of more tables. A view derives its data from the tables on which it is based. These tables are called base tables. Views can be based on actual tables or another view also.

Views are very powerful and handy since they can be treated just like any other table but do not occupy the space of a table.

Creating Views: Suppose we have EMP and DEPT table. To see the empno, ename, sal, deptno, department name and location we have to give a join query like this.

```
select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc

    from emp e, dept d where e.deptno=d.deptno;
```

Update a View: If a view is based on a single underlying table, you can insert, update, or delete rows in the view. This will actually insert, update, or delete rows in the underlying table.

There are restrictions on your ability to do this, although the restrictions are quite sensible:

- You cannot insert if the underlying table has any NOT NULL columns that don't appear in the view.
- You cannot insert or update if any one of the view's columns referenced in the insert or update contains functions or calculations.
- You cannot insert, update, or delete if the view contains group by, distinct, or a reference to the pseudo-column RowNum.

Stability of a View.
If the base table is dropped then the view becomes invalid.

```
create view RAIN_VIEW as
select City, Precipitation
from TROUBLE;

View created.

drop table TROUBLE;
select * from RAIN_VIEW;

 *
ERROR at line 1:
ORA-04063: view "PRACTICE.RAIN_VIEW" has errors


create or replace view RAIN_VIEW as
select * from TROUBLE;

alter table TROUBLE add (Warning
 VARCHAR2(20)
);
Table altered.


 Despite the change to the view's base table, the view is still valid,
 but the Warning column is missing.
```

Creating a Read-Only View. Use the *with read only* clause.

```
 create or replace view RAIN_READ_ONLY as
select * from TROUBLE
with read only;
```

Introduction to Database Triggers: Database triggers are specialized stored programs. As such, they are defined by very similar DDL rules. Likewise, triggers can call SQL statements and PL/SQL functions and procedures. You can choose to implement triggers in PL/SQL or Java.

Database triggers differ from stored functions and procedures because you can't call them directly. Database triggers are fired when a triggering event occurs in the database.

10.1.1 Data Definition Language triggers

DDL triggers

These triggers fire when you create, change, or remove objects in a database schema. They are useful to control or monitor DDL statements.

table and trigger share the same name as they use two separate namespaces.

10.1.2 Events that work with DDL triggers

(For details See Table 10-1 of Book Oracle 11g PL/SQL Programming)

ALTER,CREATE, DROP, GRANT, RENAME, REVOKE.

10.1.3 Event Attribute Functions

These are system-defined event attribute functions. For instance:

```
ORA_DICT_OBJ_NAME
ORA_DICT_OBJ_OWNER
ORA_DICT_OBJ_TYPE
ORA_CLIENT_IP_ADDRESS
ORA_DATABASE_NAME
ORA_IS_ALTER_COLUMN
ORA_LOGIN_USER
SPACE_ERROR_INFO
ORA_SYSEVENT
```

How to use them: Just like any other function.
Example:

```
 DECLARE
password VARCHAR2(60);
BEGIN
IF ora_dict_obj_type = 'USER' THEN
password := ora_des_encrypted_password;
END IF;
END;
```

```
DECLARE
ip_address VARCHAR2(11);
BEGIN
IF ora_sysevent = 'LOGON' THEN
ip_address := ora_client_ip_address;
```

```
END IF;
END;
```

10.1.4 Building DDL Triggers

Syntax:

```
 CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} ddl_event ON {DATABASE | SCHEMA}
[WHEN (logical_expression)]
[DECLARE]
declaration_statements;
BEGIN
execution_statements;
END [trigger_name];
/
```

Few Points:

- **Schema trigger**: The keyword schema decides its type. A SCHEMA trigger is created on a schema and fires whenever the user who owns it is the current user and initiates the triggering event.

  Suppose that both user1 and user2 own schema triggers, and user1 invokes a DR unit owned by user2. Inside the DR unit, user2 is the current user. Therefore, if the DR unit initiates the triggering event of a schema trigger that user2 owns, then that trigger fires. However, if the DR unit initiates the triggering event of a schema trigger that user1 owns, then that trigger does not fire. [present a suitable example in this regard.]

- **DATABASE Triggers:** Use keyword DATABASE for it. A DATABASE trigger is created on the database and fires whenever any database user initiates the triggering event.

Example Suppose we want to log 3 types of DDL (CREATE,ALTER AND DELETE) for a particular user (schema).

Note: Event Attribute Functions have been used to get the necessary information.

```
create or replace trigger ddl_trigger
    before create or alter or drop on SCHEMA
    begin
    dbms_output.put_line('Who did it?  '||ora_dict_obj_owner);
        dbms_output.put_line('What was the Operation?  ' ||ora_sysevent);
        dbms_output.put_line('On what?  ' ||ora_dict_obj_name);
        dbms_output.put_line('On type of object it was?  ' ||ora_dict_obj_type);
    end;
```

10.1.5 Data Manipulation Language Triggers

DML triggers can fire before or after INSERT, UPDATE, and DELETE statements.

Row-level and statement-level trigger: DML triggers can be statement- or row-level activities. Statement-level triggers fire and perform a statement or set of statements once no matter how many rows are affected by the DML event. Row-level triggers fire and perform a statement or set of statements for each row changed by a DML statement.

DML triggers Syntax:

```
 CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER}
{INSERT | UPDATE | UPDATE OF column1 [, column2 [, column(n+1)]] | DELETE}
ON table_name
[FOR EACH ROW]
[WHEN (logical_expression)]
[DECLARE]
[PRAGMA AUTONOMOUS_TRANSACTION;]
declaration_statements;
BEGIN
execution_statements;
END [trigger_name];
```

Note: The clause *[FOR EACH ROW]* specifies that it is row-level trigger.

Example:

```
 CREATE OR REPLACE TRIGGER demo_trigger_types1
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  --FOR EACH ROW

--DECLARE

BEGIN

    dbms_output.put_line('Row Level Trigger Fires each time. ');

END;
```

```
 CREATE OR REPLACE TRIGGER demo_trigger_types1
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  --FOR EACH ROW
```

```
--DECLARE

BEGIN

    dbms_output.put_line('Row Level Trigger Fires . ');

END;
```

10.1.6 OLD and NEW Pseudorecords

When a row-level trigger fires, the PL/SQL runtime system creates and populates the two pseudorecords OLD and NEW. They are called pseudorecords because they have some, but not all, of the properties of records.

For the row that the trigger is processing:

- For an INSERT trigger, OLD contains no values, and NEW contains the new values.
- For an UPDATE trigger, OLD contains the old values, and NEW contains the new values.
- For a DELETE trigger, OLD contains the old values, and NEW contains no values.

Example:

```
  Example 2: Need to discuss OLD and NEW

CREATE OR REPLACE TRIGGER demo_old_new
  BEFORE UPDATE ON employees
  FOR EACH ROW
WHEN (NEW.EMPLOYEE_ID > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff  := :NEW.SALARY  - :OLD.SALARY;
    dbms_output.put('Old salary: ' || :OLD.SALARY);
    dbms_output.put('  New salary: ' || :NEW.SALARY);
    dbms_output.put_line('  Difference ' || sal_diff);
END;
```

```
--Another example-

CREATE OR REPLACE TRIGGER EVAL_CHANGE_TRIGGER
  AFTER INSERT OR UPDATE OR DELETE
  ON EVALUATIONS
DECLARE
  log_action  EVALUATIONS_LOG.action%TYPE;
BEGIN
  IF INSERTING THEN
    log_action := 'Insert';
  ELSIF UPDATING THEN
    log_action := 'Update';
  ELSIF DELETING THEN
    log_action := 'Delete';
  ELSE
    DBMS_OUTPUT.PUT_LINE('This code is not reachable.');
  END IF;

  INSERT INTO EVALUATIONS_LOG (log_date, action)
    VALUES (SYSDATE, log_action);
END;
```

Side Note: Create Sequence. Mainly used to handle an autonumber field.

```
Syntax:
CREATE SEQUENCE sequence_name
MINVALUE value
MAXVALUE value
START WITH value
INCREMENT BY value
CACHE value;


--Example--
CREATE SEQUENCE supplier_seq
MINVALUE 1
MAXVALUE 999999
START WITH 1
INCREMENT BY 1
CACHE 20;
```

```
  ---How to use it--
  supplier_seq.NEXTVAL;
```

New and Old Example:

```
  CREATE OR REPLACE
TRIGGER NEW_EVALUATION_TRIGGER
BEFORE INSERT ON EVALUATIONS
FOR EACH ROW
BEGIN
  :NEW.evaluation_id := evaluations_sequence.NEXTVAL
END;
```

Another example from Book: Use Regular Expression.

The following example demonstrates a trigger that replaces a whitespace in a last name with a dash for hyphenated names:

```
 CREATE OR REPLACE TRIGGER TRG_REG
BEFORE INSERT ON EMP
FOR EACH ROW
WHEN (REGEXP_LIKE(NEW.name,' '))
BEGIN
  :NEW.name := REGEXP_REPLACE(:NEW.Name,' ','-',1,1);
END TRG_REG;
```

NOTE: See the format of NEW Pseudorecord inside hte WHEN clause. There is no : before it.

10.1.7 Trigger Enhancements (11g)

Resource Extracted from: http://oracle-base.com/articles/11g/trigger-enhancements-11gr1.php#compound_triggers.

10.1.7.1 Order of Firing Trigger

Oracle allows more than one trigger to be created *for the same timing point*, but it has *never guaranteed the execution order of those triggers.* The *Oracle 11g* trigger syntax now

includes the *FOLLOWS* clause to guarantee execution order for triggers defined with the same timing point. The following example creates a table with two triggers for the same timing point.

```
  CREATE TABLE trigger_follows_test (
  id          NUMBER,
  description VARCHAR2(50)
);

CREATE OR REPLACE TRIGGER trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_1 - Executed');
END;
/

CREATE OR REPLACE TRIGGER trigger_follows_test_trg_2
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_2 - Executed');
END;
/
```

If we insert into the test table, there is no guarantee of the execution order.

```
  SQL> SET SERVEROUTPUT ON
SQL> INSERT INTO trigger_follows_test VALUES (1, 'ONE');
TRIGGER_FOLLOWS_TEST_TRG_1 - Executed
TRIGGER_FOLLOWS_TEST_TRG_2 - Executed

1 row created.

SQL>
```

Now our objective is:
We can specify that the `TRIGGER_FOLLOWS_TEST_TRG_2` **trigger should be executed before the** `TRIGGER_FOLLOWS_TEST_TRG_1` trigger by recreating the `TRIGGER_FOLLOWS_TEST_TRG_1` trigger using the FOLLOWS clause.

```
  CREATE OR REPLACE TRIGGER trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
FOLLOWS trigger_follows_test_trg_2
BEGIN
  DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_1 - Executed');
END;
/
```

Now the `TRIGGER_FOLLOWS_TEST_TRG_1` trigger always follows the `TRIGGER_FOLLOWS_TEST_TRG_2` trigger.

```
  SQL> SET SERVEROUTPUT ON
SQL> INSERT INTO trigger_follows_test VALUES (2, 'TWO');
TRIGGER_FOLLOWS_TEST_TRG_2 - Executed
TRIGGER_FOLLOWS_TEST_TRG_1 - Executed

1 row created.

SQL>
```

10.1.7.2 Compound Triggers

A compound trigger allows code for one or more timing points for a specific object to be combined into a single trigger. The individual timing points can share a single global declaration section, whose state is maintained for the lifetime of the statement. Once a statement ends, due to successful completion or an error, the trigger state is cleaned up. In previous releases this type of functionality was only possible by defining multiple triggers whose code and global variables were defined in a separate package, as shown in the Mutating Table Exceptions article, but the compound trigger allows for a much tidier solution.

The triggering actions are defined in the same way as any other DML trigger, with the addition of the COMPOUND TRIGGER clause. The main body of the trigger is made up of an optional global declaration section and one or more timing point sections, each of which may contain a local declaration section whose state is not maintained.

See the example in the web given at the beginning of the section.

10.1.7.3 Enable and Disable Triggers

It has been possible to enable and disable triggers for some time using the ALTER TRIGGER and ALTER TABLE commands.

```
ALTER TRIGGER <trigger-name> DISABLE;

ALTER TRIGGER <trigger-name> ENABLE;

ALTER TABLE <table-name> DISABLE ALL TRIGGERS;

ALTER TABLE <table-name> ENABLE ALL TRIGGERS;
```

Chapter 11

Lecture (Lec 19 & 20) for Week 12

11.1 Collections

A collection is a data structure that acts like a list or a single-dimensional array.
Associative Array and VARRAY.

11.1.1 Associative Array or Index-By Table

An index-by table (also called an associative array) is a set of **key-value** pairs. **Each key is unique** and is used to locate the corresponding value. The key can be either an integer or a string.

Example Code:

```
DECLARE
TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
salary_list salary;
name   VARCHAR2(20);
BEGIN
-- adding elements to the table
salary_list('A. Rahim') := 62000;
salary_list('A. Karim') := 75000;
salary_list('Martin') := 100000;
salary_list('James') := 75000;

-- printing the table
name := salary_list.FIRST;
WHILE name IS NOT null LOOP
dbms_output.put_line
('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
name := salary_list.NEXT(name);
END LOOP;
END;
```

```
/

--output---

Salary of A. Karim is 75000
Salary of A. Rahim is 62000
Salary of James is 75000
Salary of Martin is 100000

PL/SQL procedure successfully completed.
```

Example Code 2

```
DECLARE
TYPE list_of_names_t IS TABLE OF varchar2(100)
INDEX BY PLS_INTEGER;
happyfamily list_of_names_t;
l_row PLS_INTEGER;
BEGIN
happyfamily (2020202020) := 'Eli';
happyfamily (-15070) := 'Steven';
happyfamily (-90900) := 'Chris';
happyfamily (88) := 'Veva';
happyfamily(1):='Raihan';
l_row := happyfamily.FIRST;


WHILE (l_row is not null)
loop
dbms_output.put_line('OUTPUT shown:'|| happyfamily(l_row));
l_row:=happyfamily.NEXT(l_row);
end loop;

dbms_output.put_line('Total no is:'||happyfamily.count);
END;
```

11.1.2 VARRAYS

Varrays were introduced in Oracle 8. They are densely populated arrays and behave like traditional programming arrays. They may be stored in permanent tables and accessed by SQL. At creation, they have a fixed size that cannot change.

Declare a VARRAY type: To declare a VARRAY type, you use this syntax:

```
TYPE type_name IS VARRAY(max_elements)
OF element_type [NOT NULL];
```

In this declaration:

- `type\_name` is the type of the VARRAY.
- `max_elements` is the maximum number of elements allowed in the VARRAY.
- NOT NULL specifies that the element of the VARRAY of that type cannot have NULL elements. Note that a VARRAY variable can be null, or uninitialized.
- `element_type` is the type of elements of the VARRAY type's variable.

Initialize VARRAY variables

To initialize a VARRAY variable to an empty collection (zero elements), you use the following syntax:

```
varray_name type_name := type_name();
```

If you want to specify elements for the VARRAY variable while initializing it, you can use this syntax:

```
varray_name type_name := type_name(element1, element2, ...);
```

Accessing array elements:

```
varray_name(n);
```

Example with VARRAY:

```
DECLARE
TYPE t_name_type IS VARRAY(2)
OF VARCHAR2(20) NOT NULL;
t_names t_name_type  := t_name_type('John','Jane');
t_enames t_name_type := t_name_type();
cnt number:=0;
BEGIN
-- initialize to an empty array
dbms_output.put_line('The number of elements in t_enames ' || t_enames.COUNT);

-- initialize to an array of a elements
```

```
dbms_output.put_line('The number of elements in t_names ' || t_names.COUNT);

----traverse the entire array
cnt:=t_names.count;
for i in 1..cnt
loop
dbms_output.put_line( t_names(i));
end loop;
END;
/


---output--

The number of elements in t_enames 0
The number of elements in t_names 2
John
Jane

PL/SQL procedure successfully complete
```

Example 2:

```
DECLARE
type namesarray IS VARRAY(5) OF VARCHAR2(10);
type grades IS VARRAY(5) OF INTEGER;
names namesarray;
marks grades;
total integer;
BEGIN
names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
marks:= grades(98, 97, 78, 87, 92);
total := names.count;
dbms_output.put_line('Total '|| total || ' Students');
FOR i in 1 .. total LOOP
dbms_output.put_line('Student: ' || names(i) || '
Marks: ' || marks(i));
END LOOP;
END;
/
```

`n` is the index of the element, which **begins with 1** and ends with the `max_elements` the maximum number of elements defined in the VARRAY type.

VARRAY in a Table structure:

```
CREATE OR REPLACE TYPE address_varray
AS VARRAY(3) OF VARCHAR2(30 CHAR);
/

CREATE TABLE addresses
(address_id INTEGER NOT NULL
,individual_id INTEGER NOT NULL
,street_address ADDRESS_VARRAY NOT NULL
,city VARCHAR2(20 CHAR) NOT NULL
,state VARCHAR2(20 CHAR) NOT NULL
,postal_code VARCHAR2(20 CHAR) NOT NULL
,country_code VARCHAR2(10 CHAR) NOT NULL
,CONSTRAINT address_pk PRIMARY KEY (address_id)
);
```

```
INSERT
INTO addresses VALUES
(11
,11
,address_varray
('Office of Senator McCain'
,'450 West Paseo Redondo'
,'Suite 200')
,'Tucson'
,'AZ'
,'85701'
,'USA');
```

The example program inserts a full set of three rows into the varray data type. It is important to note that in the values clause, the varray data type name is used as the **constructor name**.