Experiment no: - 9

Experiment Name: - Trigger

Aim: - Performing practical by using trigger concept.

Resource required: - Oracle 9i - iSQLplus

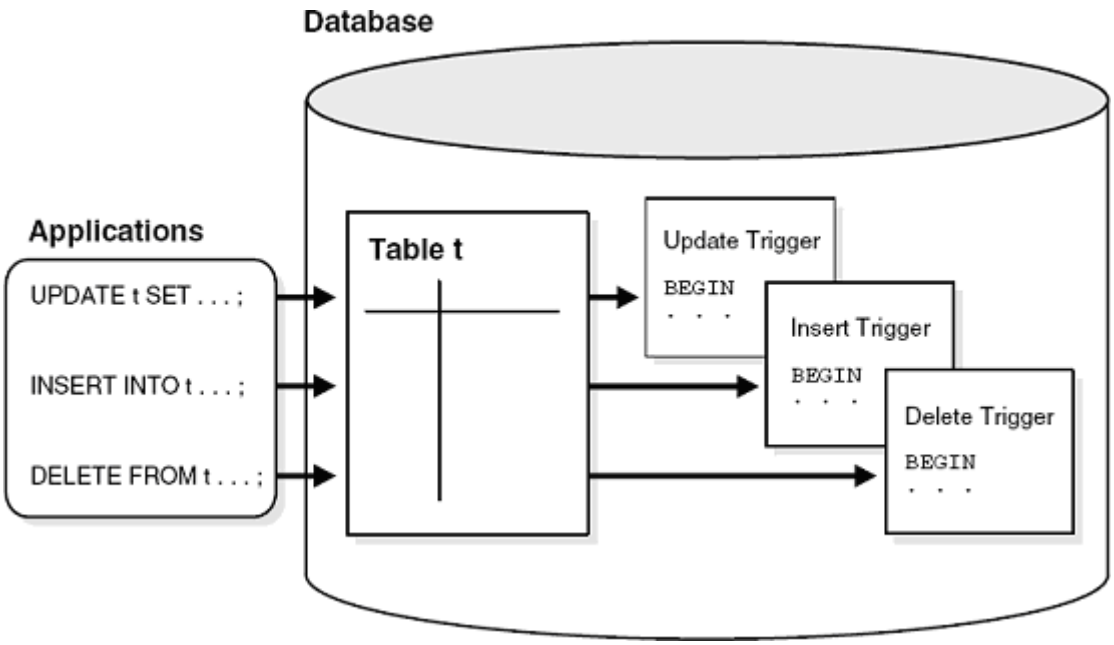
Theory: -

- **TRIGGER:**

- Triggers are similar to stored procedures.

- 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.

- Triggers are implicitly fired by Oracle when a triggering event occurs, no matter which user is connected or which application is being used.



Syntax:

```
Create Trigger <trigger name>
        (AFTER / BEFORE) <triggering events> ON <table name>
        [ FOR EACH ROW ]
        [ WHEN < condition> ]
        < trigger actions >;
        < triggering events> :: = < trigger event> { OR <trigger event>}
        < trigger event> :: = Insert/ Delete/ Update [ OF <column name >
                                                        {, <column name> } ]
        < trigger action > :: = < PL/SQL block>
```

Types of Triggers

When you define a trigger, you can specify the number of times the trigger action is to be run:

- Once for every row affected by the triggering statement, such as a trigger fired by an UPDATE statement that updates many rows
- Once for the triggering statement, no matter how many rows it affects

➢ Row Triggers:

A **row 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 run.

➢ Statement Triggers:

A **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.

➢ BEFORE and AFTER Triggers:

When defining a trigger, you can specify the **trigger timing**--whether the trigger action is to be run before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers

Insert Triggers:

A BEFORE OR AFTER INSERT Trigger means that Oracle will fire this trigger before or after the INSERT operation is executed.

Syntax:

```
CREATE or REPLACE TRIGGER trigger_name
BEFORE OR AFTER INSERT
   ON < table name>
   [FOR EACH ROW]
DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```

Update Triggers:

A BEFORE OR AFTER UPDATE Trigger means that Oracle will fire this trigger before the UPDATE operation is executed.

Syntax:

```
CREATE or REPLACE TRIGGER trigger_name
BEFORE OR AFTER UPDATE
   ON table_name
   [ FOR EACH ROW ]
DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```

Delete Triggers: A BEFORE OR AFTER DELETE Trigger means that Oracle will fire this trigger before or after the DELETE operation is executed

Syntax:

```
CREATE or REPLACE TRIGGER trigger_name
BEFORE OR AFTER DELETE
   ON table_name
   [ FOR EACH ROW ]
DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```

Drop Triggers:

Syntax:

```
DROP TRIGGER trigger_name;
```

SAMPLE EXAMPLES:

Consider Relation/Table/Entity: Employees.
Following question for practice. And Students are required to write the output.

Create a table as follows:

```
CREATE TABLE orders
( order_id          number(5),
```

```
quantity         number(4),
cost_per_item number(6,2),
total_cost         number(8,2),
create_date        date,
created_by        varchar2(10);
);
```

Q:
```
CREATE VIEW empvu80
    AS SELECT    employee_id, last-name, salary
    FROM  employees
   WHERE DEPARTMENT_ID=80;
```
Output:

Q:
```
SELECT *
    FROM empvu80;
```
Output:

Q:
```
SELECT ROWNUM as RANK, last_name, salary
        FROM (SELECT   last_name, salary   FROM    employees
       ORDER BY salary  DECS)
       WHERE    RONUM <= 3;
```
Output:

Q:
```
CREATE OR REPLACE TRIGGER orders_before_insert
  BEFORE INSERT
 ON orders
 FOR EACH ROW

    DECLARE
      v_username varchar2 (10);
    BEGIN
      -- Find username of person performing INSERT into table
      SELECT user INTO v_username
      FROM dual;
      -- Update created_by field to the username of the person performing the
    INSERT
      : new.created_by:= v_username;
    END;
```
Output:

Q:
```
CREATE OR REPLACE TRIGGER orders_after_update
  AFTER UPDATE
```

```
   ON orders
   FOR EACH ROW

  DECLARE
     v_username varchar2 (10);
  BEGIN
            -- Find username of person performing UPDATE into table
            SELECT user INTO v_username
            FROM dual;
            -- Insert record into audit table
            INSERT INTO orders_audit
             ( order_id, quantity_before, quantity_after, username )
            VALUES
             ( :new.order_id, :old.quantity, :new.quantity, v_username );
      END;
```

Output:

Q: CREATE OR REPLACE TRIGGER orders_after_delete

```
    AFTER DELETE
    ON orders
    FOR EACH ROW

  DECLARE
      v_username varchar2 (10);
  BEGIN
            -- Find username of person performing the DELETE on the table
            SELECT user INTO v_username
            FROM dual;
            -- Insert record into audit table
            INSERT INTO orders_audit
             ( order_id, quantity, cost_per_item, total_cost, delete_date, deleted_by)
            VALUES
             ( :old.order_id, :old.quantity, :old.cost_per_item, :old.total_cost,
               sysdate, v_username );
    END;
```

Output:

Q**:** DROP TRIGGER orders_before_insert;

Output:

Conclusion:

In this practical, learned how to create view and implement triggering concept on database object.

LAB ASSIGNMENT -9

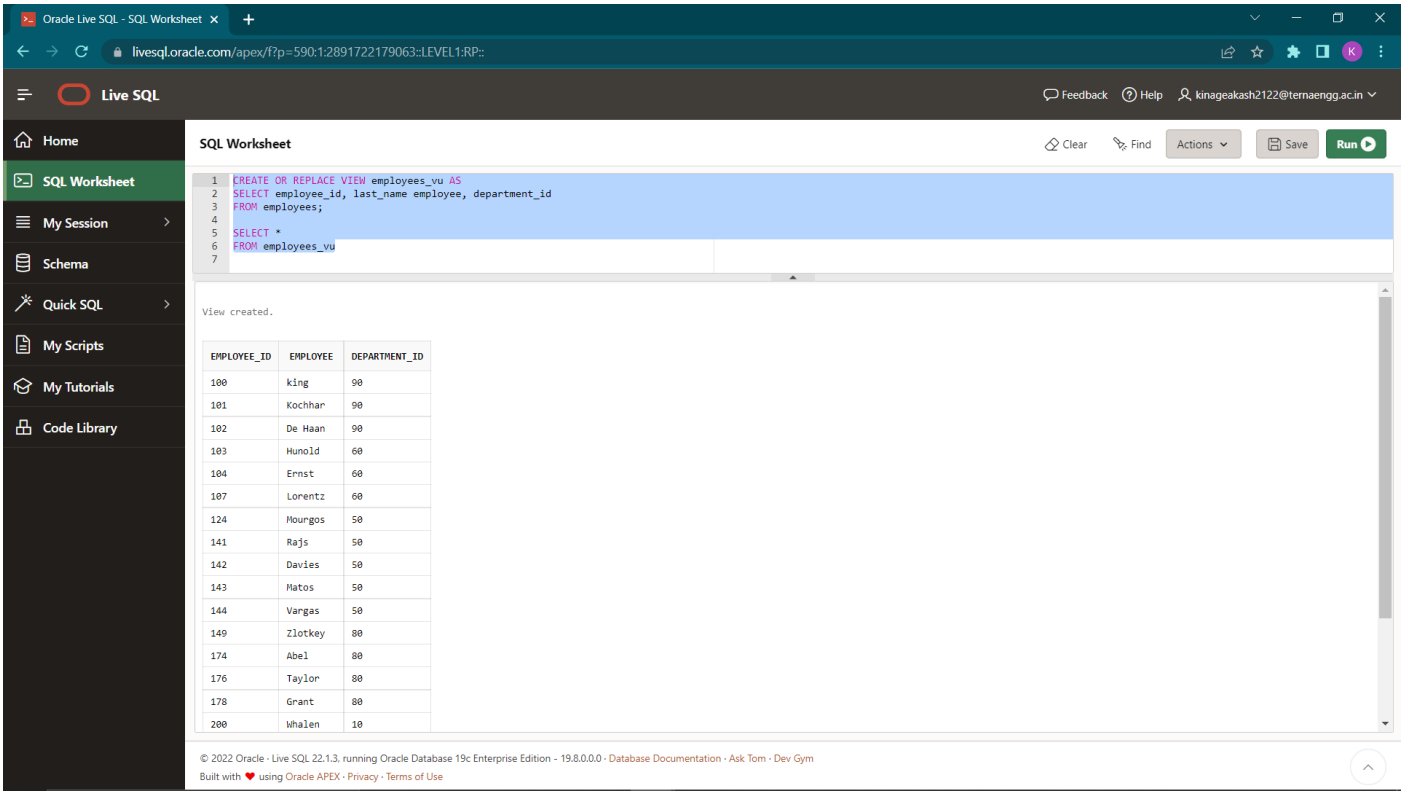
1. Create a view called EMPLOYEE_VU based on the employee numbers, employee names, and department numbers from the EMPLOYEE table. Changes the heading for the employee name to EMPLOYEE. And Display the contents of the EMPLOYEE_VU view.

Query :-

```
CREATE OR REPLACE VIEW employees_vu AS
SELECT employee_id, last_name employee, department_id
FROM employees;

SELECT *
FROM employees_vu
```

Output :-

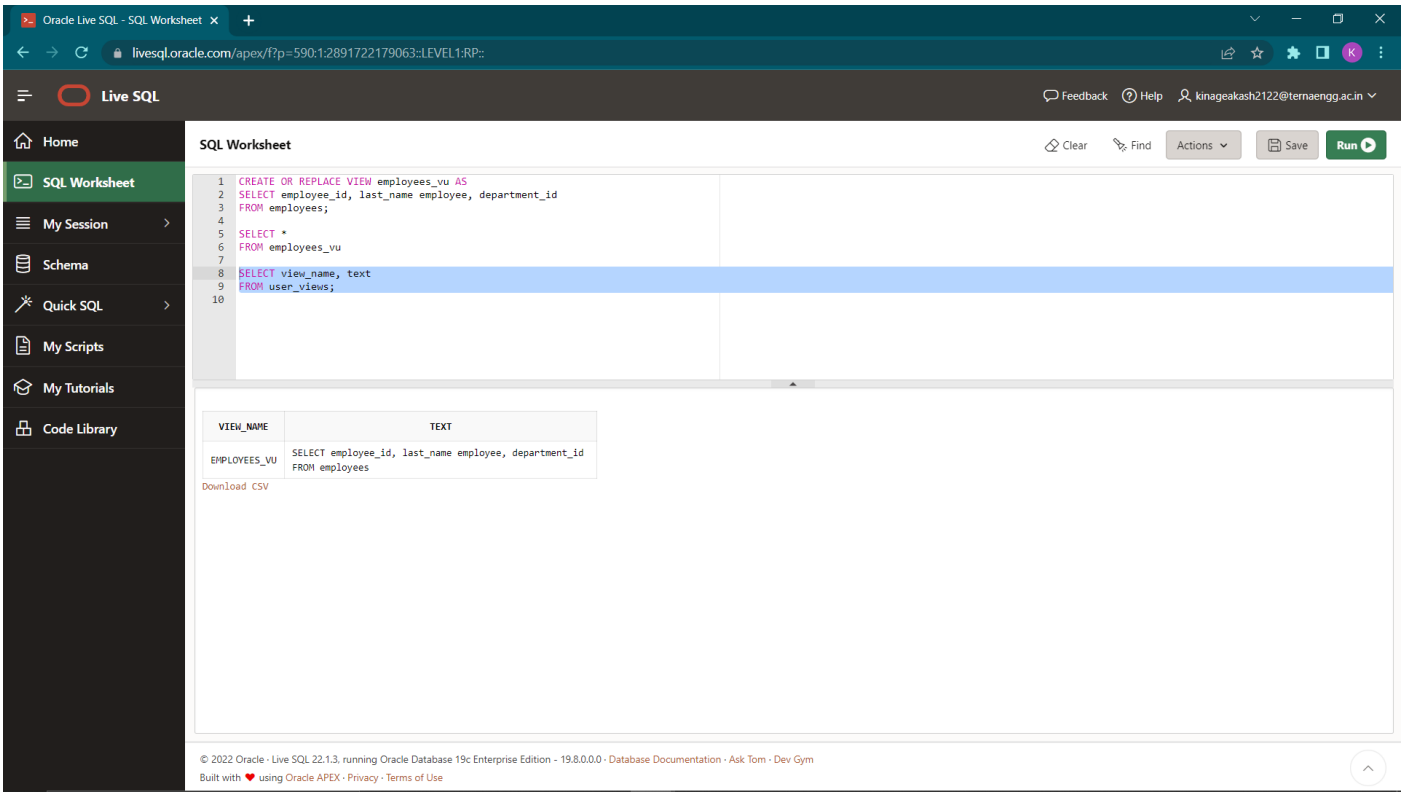


2. Select the view name and text from the USER_VIEWS data dictionary view.

Query :-

```
SELECT view_name, text
FROM user_views;
```

Output :-

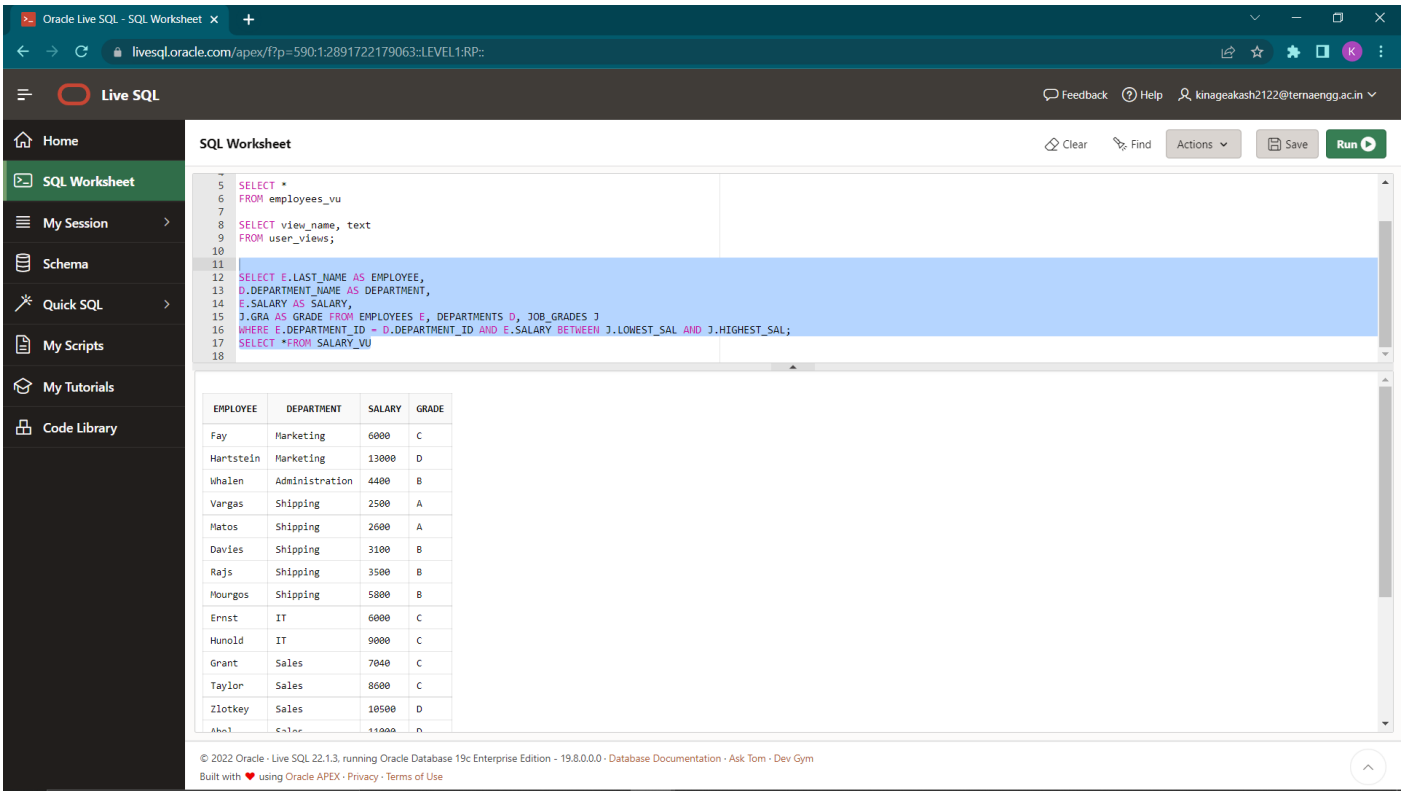


3. Create a view called SALARY_VU based on the employee last names, department names, salaries, and salary grades for all employees. Use the EMPLOYEES, DEPARTMENTS, and JOB_GRADES tables. Label the column Employee, Department, Salary, and Grade, respectively.

Query :-

```
SELECT E.LAST_NAME AS EMPLOYEE,
D.DEPARTMENT_NAME AS DEPARTMENT,
E.SALARY AS SALARY,
J.GRA AS GRADE FROM EMPLOYEES E, DEPARTMENTS D, JOB_GRADES J
WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID AND E.SALARY BETWEEN J.LOWEST_SAL
AND J.HIGHEST_SAL;
SELECT *FROM SALARY_VU
```

Output :-



4. Create a table emp (eno, ename, hrs, pno, super_no) and project (pname, pno, thrs, head_no) where thrs is the total hours and is the derived

attribute. Its value is the sum of hrs of all employees working on that project. eno and pno are primary keys, head_no is foreign key to emp relation. Insert 4 tuples and write triggers to do the following:

a) Creating a trigger to insert a new employee tuple and display the new total hours from project table.

b) Creating a trigger to change the project of an employee and display the new total hours from project table.

c) Creating a trigger to deleting the project of an employee.