`

**Batch: A3 Roll No.: 16010123066**

**Experiment No. 5**

|  |
| --- |
| **Title: Queries based Views and Triggers** |



**Objective:** To be able to use SQL view and triggers.



**Expected Outcome of Experiment:**

CO3: Utilize SQL for Relational Database Operations

**Books/ Journals/ Websites referred:**

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press

2. www.db-book.com

3. Korth, Slberchatz, Sudarshan : “Database Systems Concept”, 5th Edition , McGraw

Hill

4. Elmasri and Navathe,”Fundamentals of database Systems”, 4th Edition,PEARSON

Education.

**Resources used:** Postgresql

**Theory**

**View**

Views are pseudo-tables. That is, they are not real tables; nevertheless appear as ordinary tables to SELECT. A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table. A view can even represent joined tables. Because views are assigned separate permissions, you can use them to restrict table access so that the users see only specific rows or columns of a table.

A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables, which depends on the written PostgreSQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following −

* Structure data in a way that users or classes of users find natural or intuitive.
* Restrict access to the data such that a user can only see limited data instead of complete table.
* Summarize data from various tables, which can be used to generate reports.

Since views are not ordinary tables, you may not be able to execute a DELETE, INSERT, or UPDATE statement on a view. However, you can create a RULE to correct this problem of using DELETE, INSERT or UPDATE on a view.

Syntax

CREATE [TEMP | TEMPORARY] VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

WHERE [condition];

Ex:

CREATE VIEW COMPANY-VIEW AS

SELECT ID, NAME, AGE

FROM COMPANY;

Display view:

select \* from Company-View

Insert record into view:

Insert into Company-View values (123,’alpha’, 10)

select \* from Company

Dropping Views

Syntax: DROP VIEW view\_name;

**Triggers**

A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs. By using SQL triggers, developers can automate tasks, ensure data consistency, and keep accurate records of database activities.

For example, a trigger can be invoked when a row is inserted into a specified table or wwhen specific table columns are updated.

**Key Features of SQL Triggers:**

Automatic Execution: Triggers fire automatically when the defined event occurs (e.g., INSERT, UPDATE, DELETE).

Event-Driven: Triggers are tied to specific events that take place within the database.

Table Association: A trigger is linked to a specific table or view, and operates whenever changes are made to the table’s data.

The **basic syntax of creating a trigger** is as follows −

CREATE TRIGGER trigger\_name [BEFORE|AFTER|INSTEAD OF] event\_name

ON table\_name

[

-- Trigger logic goes here....

];

* **trigger\_name:** The name of the trigger to be created.
* **BEFORE | AFTER:** Specifies whether the trigger is fired before or after the triggering event (INSERT, UPDATE, DELETE).
* **{INSERT | UPDATE | DELETE}**: Specifies the operation that will activate the trigger.
* **table\_name:** The name of the table the trigger is associated with.
* **FOR EACH ROW:** Indicates that the trigger is row-level, meaning it executes once for each affected row. (You can optionally specify FOR EACH ROW after table name.
* )
* **trigger\_body:** The SQL statements to be executed when the trigger is fired.

The following is the syntax of creating a trigger on an UPDATE operation on one or more specified columns of a table as follows −

CREATE TRIGGER trigger\_name [BEFORE|AFTER] UPDATE OF column\_name

ON table\_name

[

-- Trigger logic goes here....

];

To use the PostgreSQL CREATE TRIGGER statement to create a trigger.

To create a new trigger in PostgreSQL, you follow these steps:

● First, create a trigger function using CREATE FUNCTION statement.

● Second, bind the trigger function to a table by using CREATE TRIGGER statement.

**Example :**

Suppose that when the name of an employee changes, you want to log it in a separate table called **employee\_audits.**

**employee**

emp\_id | fname | lname | age | salary

--------+-------+---------+-----+----------

1 | John | Doe | 30 | 50000.00

2 | Jane | Smith | 25 | 60000.00

3 | Alice | Johnson | 35 | 70000.00

(3 rows)

**Employee\_audit**

emp\_id | fname | ChangeOn |

--------+------+---------

CREATE OR REPLACE FUNCTION log\_last\_name\_changes()

RETURNS TRIGGER AS $$

BEGIN

INSERT INTO employee\_audits(employee\_id,last\_name,changed\_on)

VALUES(OLD.id,OLD.last\_name,now());

RETURN NEW;

END;

$$ language plpgsql;

CREATE TRIGGER last\_name\_changes

BEFORE UPDATE

ON employees

FOR EACH ROW

EXECUTE PROCEDURE log\_last\_name\_changes();

select \* from employee;

UPDATE employee

SET lname = 'Kulkarni'

WHERE ID = 1

select \* from semployee

select \* from employee\_audit

**Implementation Screenshots (Problem Statement, Query and Screenshots of Results):**

**VIEW:**

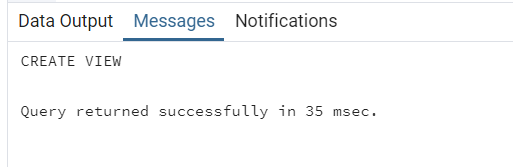
**CREATE VIEW salary\_employees AS**

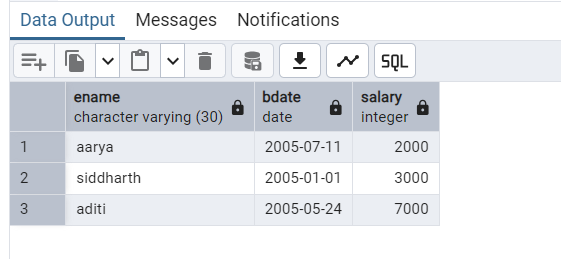
**SELECT ename, bdate, salary**

**FROM emp**

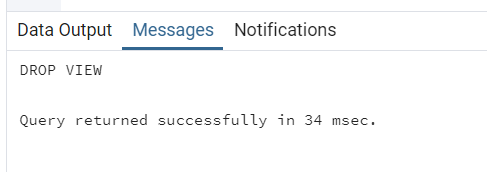
**WHERE salary > 1000;**

**SELECT \* FROM salary\_employees;**





DROP VIEW salary\_employees;



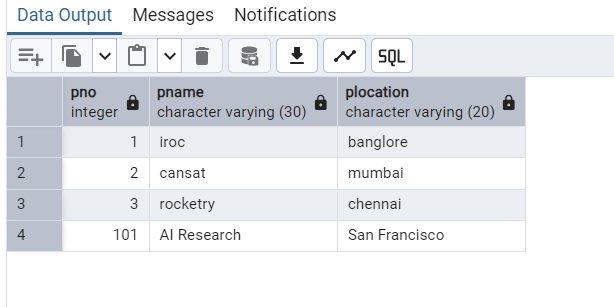
CREATE VIEW Project\_View AS

SELECT pno, pname, plocation FROM project;

INSERT INTO Project\_View (pno, pname, plocation)

VALUES (101, 'AI Research', 'San Francisco');

SELECT \* FROM Project\_View;



TRIGGER:

CREATE TABLE emp\_salary\_audit (

essn CHARACTER(9),

old\_salary INTEGER,

changed\_on TIMESTAMP

);

CREATE OR REPLACE FUNCTION log\_salary\_changes()

RETURNS TRIGGER AS $$

BEGIN

INSERT INTO emp\_salary\_audit(essn, old\_salary, changed\_on)

VALUES (OLD.essn, OLD.salary, NOW());

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER salary\_update\_trigger

BEFORE UPDATE OF salary

ON emp

FOR EACH ROW

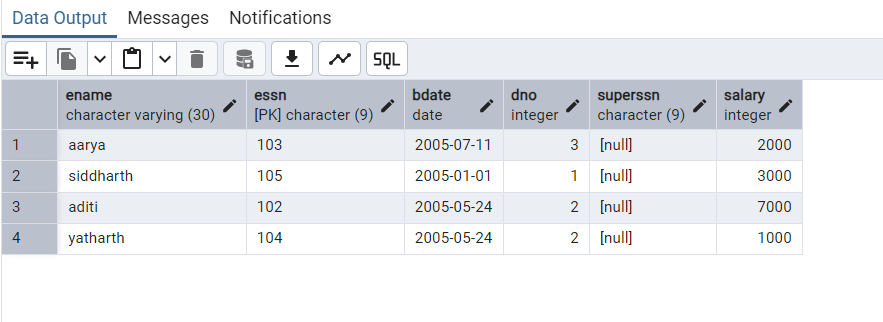
EXECUTE FUNCTION log\_salary\_changes();

SELECT \* FROM emp;

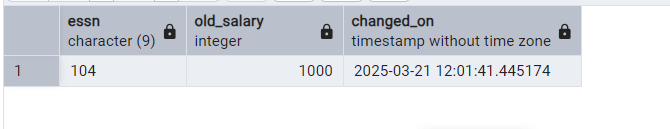
UPDATE emp

SET salary = 1000

WHERE essn = '104';



SELECT \* FROM emp\_salary\_audit;



# Conclusion:

# In this experiment, we created a view to display selected attributes from the emp table and implemented a trigger to log salary updates. This demonstrated the use of views for data abstraction and triggers for automation, ensuring data integrity and security.

# Post Lab Questions:

# What is a view in SQL, and how does it differ from a table?

A view in SQL is a virtual table that is based on the result of a SELECT query. It does not store data physically; instead, it dynamically presents data from one or more tables.

|  |  |
| --- | --- |
| **Table** | **View** |
| Stores data physically | Does not store data physically, only the query definition |
| Supports INSERT, UPDATE, DELETE directly | Can sometimes support modifications (if based on a single table without aggregations) |
| Faster, as data is directly stored | Slower, as data is retrieved dynamically from underlying tables |
| Used for storing persistent data | Used for abstraction, security, and simplifying complex queries |

# Write a query to create a view that displays only Fname, Lname, and Salary from the employee table.

CREATE VIEW emp\_salary\_view AS

SELECT Fname, Lname, salary

FROM emp;

# Write a query to drop the trigger trg\_city\_update from the employee table.

# DROP TRIGGER trg\_city\_update ON emp