**Batch: B3 Roll No.: 16010122177**

**Experiment / assignment / tutorial No. 6**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

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

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

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| CO 2 | Develop relational database design using the designed Entity-Relationship model. |
| CO 3 | Use SQL for Relational database creation, maintenance and query processing |

**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**

**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 COMPAN-VIEW AS

SELECT ID, NAME, AGE

FROM COMPANY;

select \* from Company-View

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

select \* from Company

Dropping Views

Syntax: DROP VIEW view\_name;

Triggers

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....

];

event\_name could be INSERT, DELETE, UPDATE, and TRUNCATE database operation on the mentioned table table\_name. You can optionally specify FOR EACH ROW after table name.

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....

];

Example :

creates a log table and a trigger that inserts a row in the log table after any **UPDATE** statement affects the **SALARY** column of the **EMPLOYEES** table, and then pdates **EMPLOYEES**.**SALARY** and shows the log table.

CREATE TABLE Emp\_log ( Emp\_id NUMBER, Log\_date DATE, New\_salary NUMBER, Action VARCHAR2(20));

- Create trigger that inserts row in log table after EMPLOYEES.SALARY is update

CREATE OR REPLACE TRIGGER log\_salary\_increase

AFTER UPDATE OF salary ON employees

FOR EACH ROW BEGIN INSERT INTO Emp\_log (Emp\_id, Log\_date, New\_salary, Action) VALUES (:NEW.employee\_id, SYSDATE, :NEW.salary, 'New Salary');

END;

Update EMPLOYEES.SALARY:

UPDATE employees SET salary = salary + 1000.0 WHERE Department\_id = 20;

Result:

2 rows updated. Show log table:

SELECT \* FROM Emp\_log;

Result:

EMP\_ID LOG\_DATE NEW\_SALARY ACTION

201 28-APR-10 15049.13 New Salary

202 28-APR-10 6945.75 New Salary

2 rows selected.

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

**1)Trigger**

**DROP TABLE IF EXISTS employees;**

**CREATE TABLE employees(**

**id INT GENERATED ALWAYS AS IDENTITY,**

**first\_name VARCHAR(40) NOT NULL,**

**last\_name VARCHAR(40) NOT NULL,**

**PRIMARY KEY(id)**

**);**

**CREATE TABLE employee\_audit (**

**id INT GENERATED ALWAYS AS IDENTITY,**

**employee\_id INT NOT NULL,**

**last\_name VARCHAR(40) NOT NULL,**

**changed\_on TIMESTAMP NOT NULL**

**);**

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

**RETURNS TRIGGER**

**LANGUAGE PLPGSQL**

**AS**

**$$**

**BEGIN**

**IF NEW.last\_name <> OLD.last\_name THEN**

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

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

**END IF;**

**RETURN NEW;**

**END;**

**$$;**

**CREATE TRIGGER last\_name\_changes**

**BEFORE UPDATE**

**ON employees**

**FOR EACH ROW**

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

**INSERT INTO employees (first\_name, last\_name)**

**VALUES ('John', 'Doe');**

**INSERT INTO employees (first\_name, last\_name)**

**VALUES ('Lily', 'Bush');**

**SELECT \* FROM employees;**

**UPDATE employees**

**SET last\_name = 'Brown'**

**WHERE ID = 2;**

**SELECT \* FROM employee\_audit;**

****

**Assignment**

**Code:-**

**DROP TABLE IF EXISTS details;**

**DROP TABLE IF EXISTS booking;**

**CREATE TABLE details(**

**id SERIAL PRIMARY KEY,**

**model VARCHAR(40) NOT NULL,**

**car\_type VARCHAR(40) NOT NULL**

**);**

**CREATE TABLE booking(**

**id SERIAL PRIMARY KEY,**

**booking\_id INT REFERENCES details(id),**

**booking\_status VARCHAR(40),**

**booking\_date VARCHAR(40),**

**changed\_on TIMESTAMP NOT NULL**

**);**

**CREATE OR REPLACE FUNCTION log\_details\_changes()**

**RETURNS TRIGGER**

**LANGUAGE PLPGSQL**

**AS $$**

**BEGIN**

**IF NEW.model <> OLD.model THEN**

**INSERT INTO booking(booking\_id, booking\_status, booking\_date, changed\_on)**

**VALUES(OLD.id, 'Model Changed', CURRENT\_TIMESTAMP, CURRENT\_TIMESTAMP);**

**END IF;**

**RETURN NEW;**

**END;**

**$$;**

**CREATE TRIGGER details\_changes**

**AFTER UPDATE OF model**

**ON details**

**FOR EACH ROW**

**EXECUTE PROCEDURE log\_details\_changes();**

**INSERT INTO details (model, car\_type)**

**VALUES ('swift', 'hatchback');**

**INSERT INTO details (model, car\_type)**

**VALUES ('baleno', 'sedan');**

**SELECT \* FROM details;**

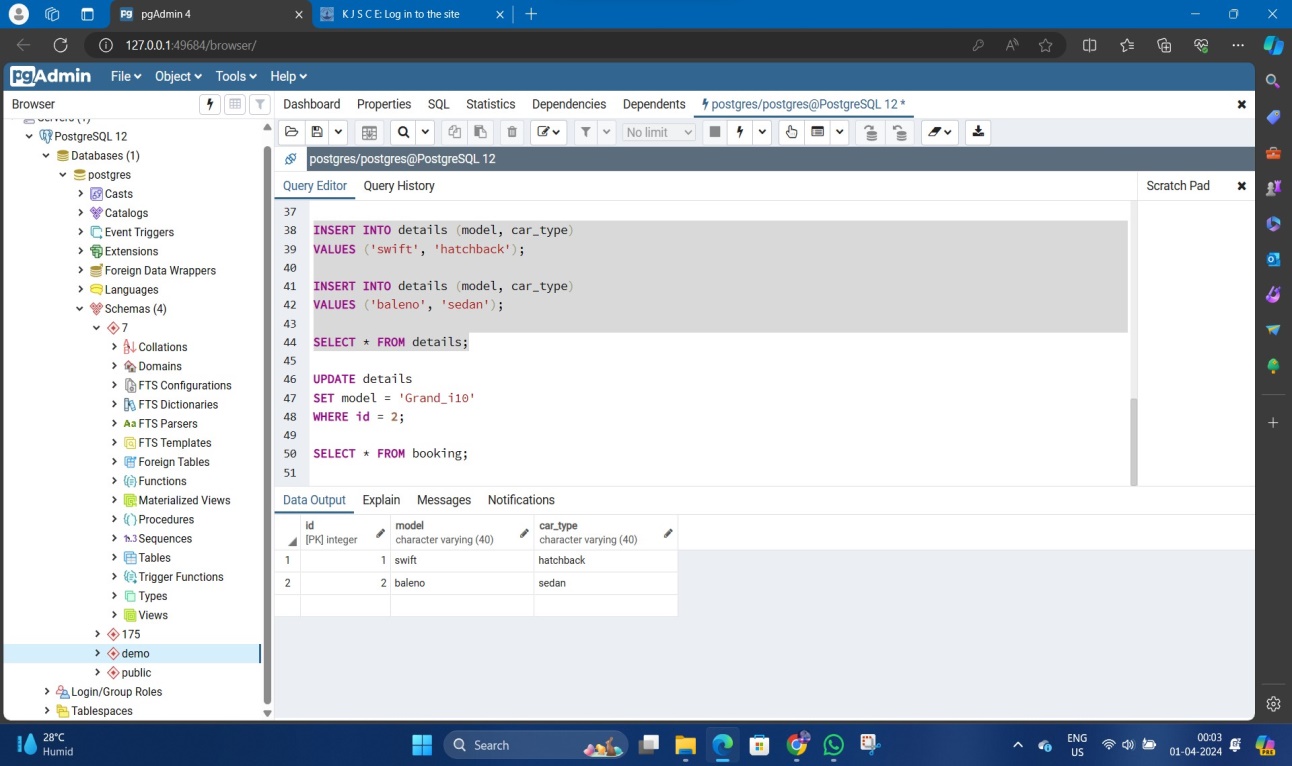
**UPDATE details**

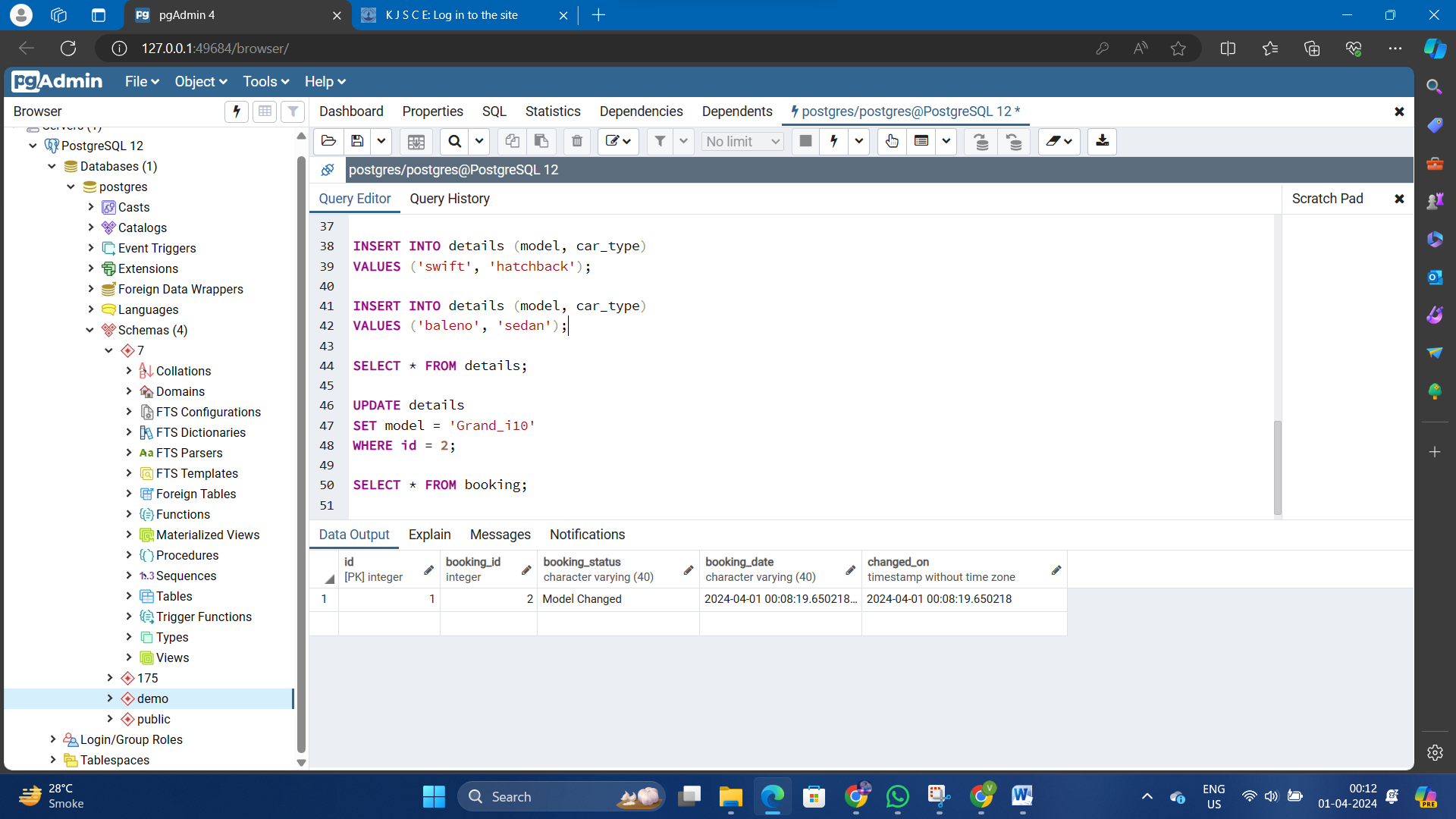
**SET model = 'Grand\_i10'**

**WHERE id = 2;**

**SELECT \* FROM booking;**

**Output:-**



****

# Conclusion:

In summary, the implementation of a trigger function within the car rental service database serves as a vital mechanism for tracking and managing changes to reservation dates. This trigger function enables the system to capture updates specifically to the reservation date column, ensuring that any modifications are meticulously recorded in an audit trail stored in the "reservation\_schedule\_audits" table. By comparing both the old and new reservation dates and appending essential details such as reservation IDs and timestamps, the trigger function enhances the overall data integrity and accountability within the car rental service system. Such a setup facilitates comprehensive auditing and analysis, empowering stakeholders to gain valuable insights into reservation scheduling dynamics while fostering transparency and reliability across the service platform.

# Post Lab Questions:

# What is a view?

# a) A view is a special stored procedure executed when certain event occurs

# b) A view is a virtual table which results of executing a pre-compiled query

# c) A view is a database diagram

# d) None of the Mentioned

# List Advantages and disadvantages of triggers

* Advantages of Triggers:

1. Data Integrity: Triggers can enforce data integrity by automatically executing actions when specific conditions are met, ensuring that only valid data is stored in the database.
2. Automated Tasks: They allow for automation of repetitive tasks, such as logging changes, updating related tables, or sending notifications, reducing the need for manual intervention.
3. Complex Constraints: Triggers enable the implementation of complex business rules or constraints that cannot be easily expressed using standard constraint mechanisms, such as foreign keys or check constraints.
4. Audit Trail: Triggers can be used to maintain an audit trail by recording changes made to the database, including who made the change and when it occurred.
5. Cross-Table Validation: Triggers can perform validation and actions across multiple tables, ensuring consistency in data modification operations.
6. Performance Improvement: In certain cases, triggers can improve performance by reducing the need for repeated manual operations, especially in scenarios where complex operations need to be performed atomically.

* Disadvantages of Triggers:

1. Implicitness: Triggers can make the behavior of the database less explicit, as actions are triggered automatically based on events, which might not be immediately obvious from looking at the schema alone.
2. Complexity: Triggers can introduce complexity to the database schema, especially if there are many triggers defined, leading to difficulties in understanding and maintaining the system.
3. Performance Overhead: Poorly designed triggers can introduce performance overhead, as they execute additional logic every time the triggering event occurs, potentially impacting the overall performance of the system.
4. Concurrency Issues: Triggers can sometimes lead to concurrency issues, such as race conditions or deadlocks, especially in scenarios involving multiple triggers or complex interactions between triggers and concurrent transactions.
5. Debugging Challenges: Triggers can make debugging and troubleshooting more challenging, as they introduce additional layers of logic that can interact with the database in unexpected ways.
6. Portability: Triggers might not be supported or might behave differently across different database management systems, limiting the portability of database schemas that rely heavily on triggers.