Batch: E-2	**Roll No.: 16010123325**
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.**

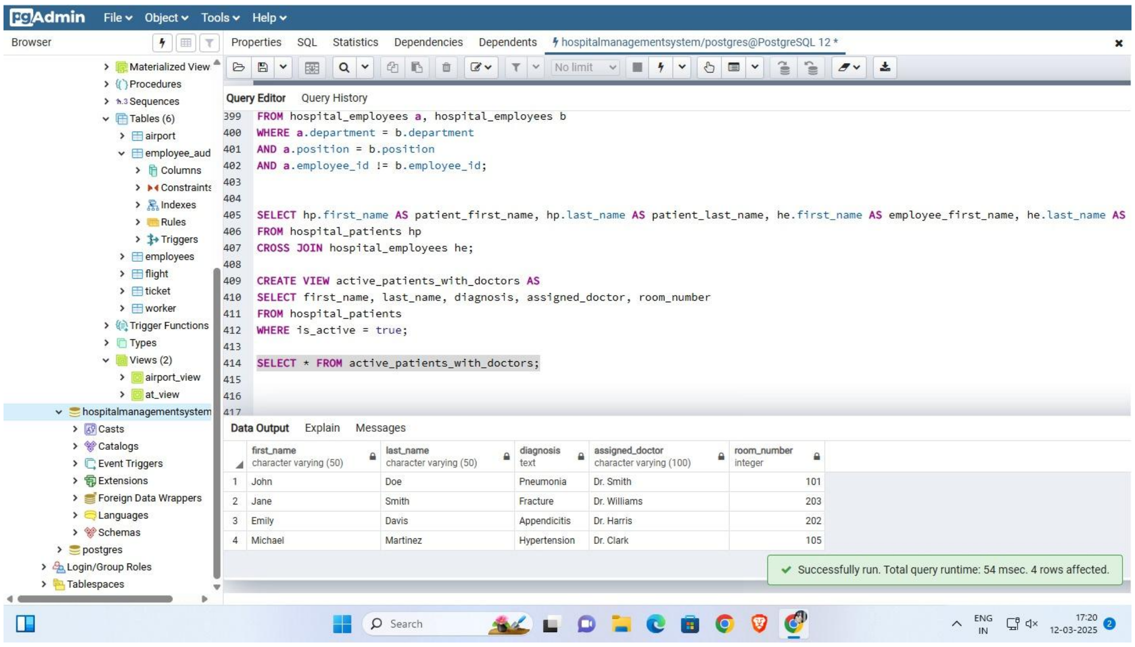
```
employee
emp_id | fname |  lname   | age |  salary
--------+-------+---------+-----+----------
      1 | John  | Doe     |  30 | 50000.00
      2 | Jane  | Smith   |  25 | 60000.00
      3 | Alice | Johnson |  35 | 70000.00
(3 rows)
Employee_audit

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):

Operations on view

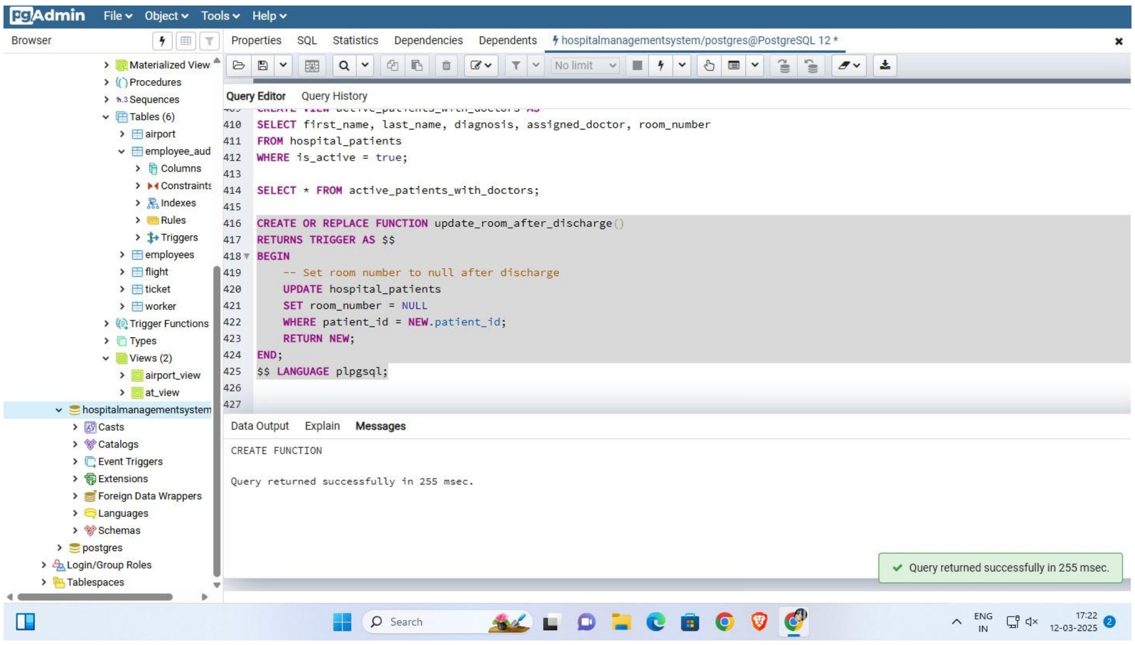
1) Selecting Active Patients with Doctors



```
FROM hospital_employees a, hospital_employees b
WHERE a.department = b.department
AND a.position = b.position
AND a.employee_id != b.employee_id;


SELECT hp.first_name AS patient_first_name, hp.last_name AS patient_last_name, he.first_name AS employee_first_name, he.last_name AS
FROM hospital_patients hp
CROSS JOIN hospital_employees he;

CREATE VIEW active_patients_with_doctors AS
SELECT first_name, last_name, diagnosis, assigned_doctor, room_number
FROM hospital_patients
WHERE is_active = true;

SELECT * FROM active_patients_with_doctors;
```

	first_name character varying (50)	last_name character varying (50)	diagnosis text	assigned_doctor character varying (100)	room_number integer
1	John	Doe	Pneumonia	Dr. Smith	101
2	Jane	Smith	Fracture	Dr. Williams	203
3	Emily	Davis	Appendicitis	Dr. Harris	202
4	Michael	Martinez	Hypertension	Dr. Clark	105

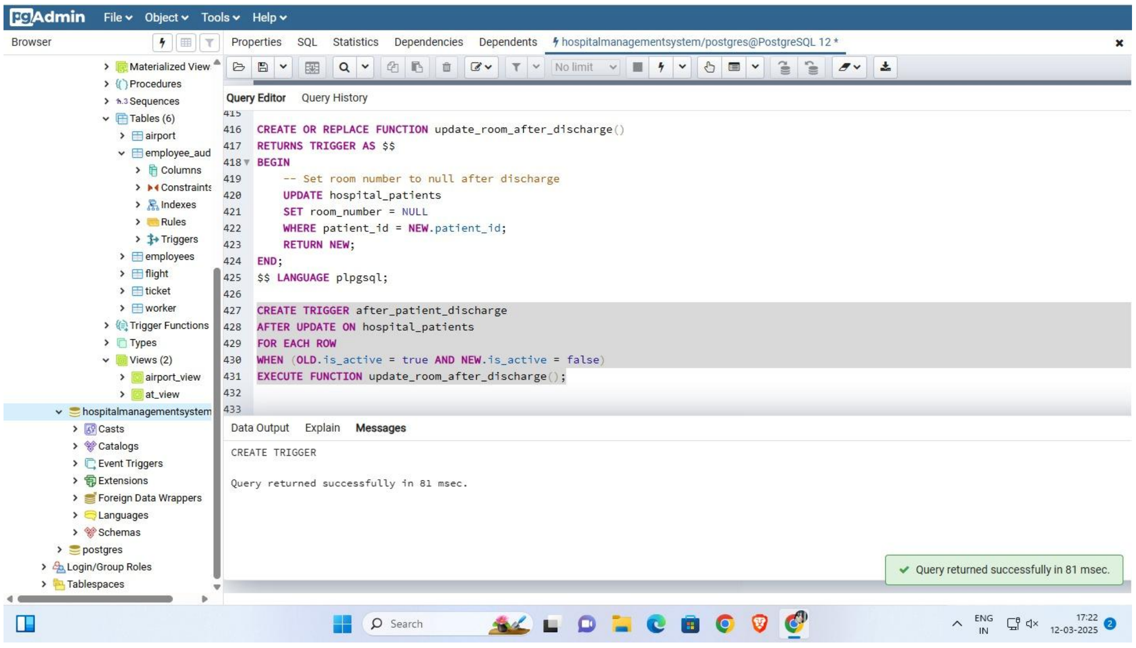
Successfully run. Total query runtime: 54 msec. 4 rows affected.



```
SELECT first_name, last_name, diagnosis, assigned_doctor, room_number
FROM hospital_patients
WHERE is_active = true;

SELECT * FROM active_patients_with_doctors;

CREATE OR REPLACE FUNCTION update_room_after_discharge()
RETURNS TRIGGER AS $$
BEGIN
    -- Set room number to null after discharge
    UPDATE hospital_patients
    SET room_number = NULL
    WHERE patient_id = NEW.patient_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

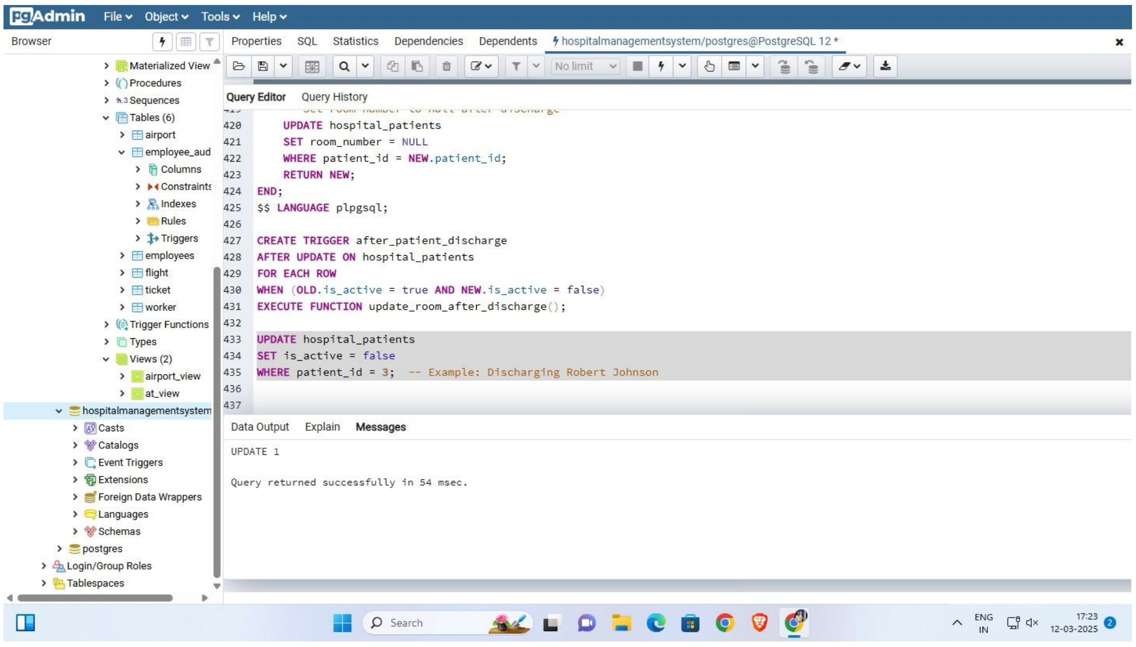
```
CREATE FUNCTION

Query returned successfully in 255 msec.
```

Query returned successfully in 255 msec.



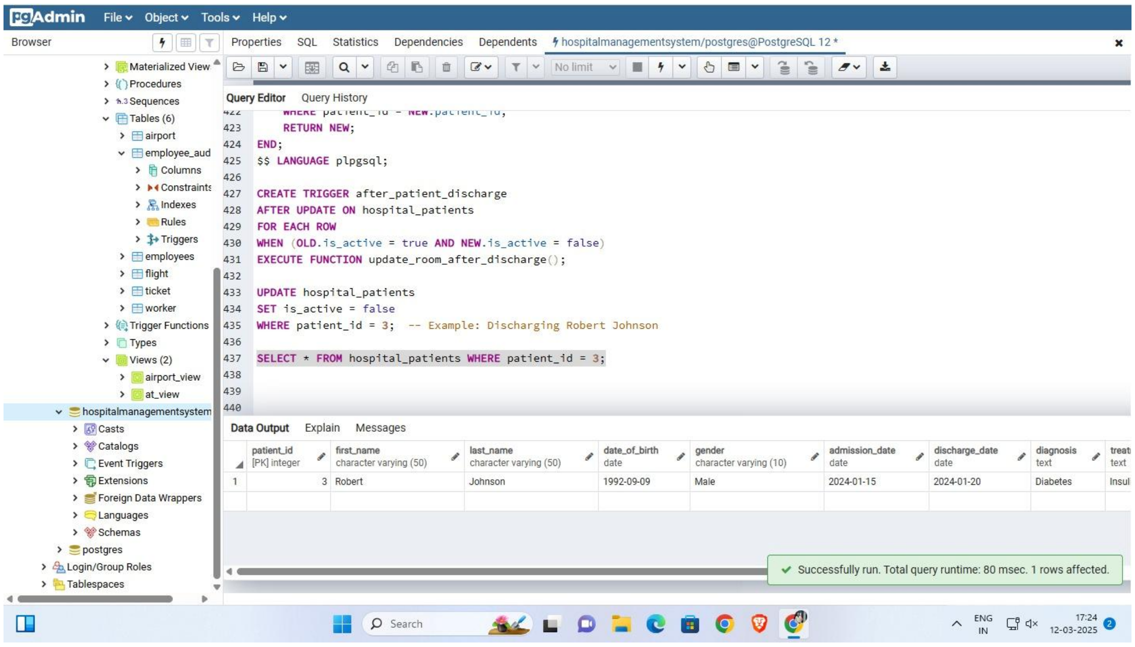
```
CREATE OR REPLACE FUNCTION update_room_after_discharge()
RETURNS TRIGGER AS $$
BEGIN
    -- Set room number to null after discharge
    UPDATE hospital_patients
    SET room_number = NULL
    WHERE patient_id = NEW.patient_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_patient_discharge
AFTER UPDATE ON hospital_patients
FOR EACH ROW
WHEN (OLD.is_active = true AND NEW.is_active = false)
EXECUTE FUNCTION update_room_after_discharge();
```

```
CREATE TRIGGER

Query returned successfully in 81 msec.
```



```
    UPDATE hospital_patients
    SET room_number = NULL
    WHERE patient_id = NEW.patient_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_patient_discharge
AFTER UPDATE ON hospital_patients
FOR EACH ROW
WHEN (OLD.is_active = true AND NEW.is_active = false)
EXECUTE FUNCTION update_room_after_discharge();

UPDATE hospital_patients
SET is_active = false
WHERE patient_id = 3;  -- Example: Discharging Robert Johnson
```

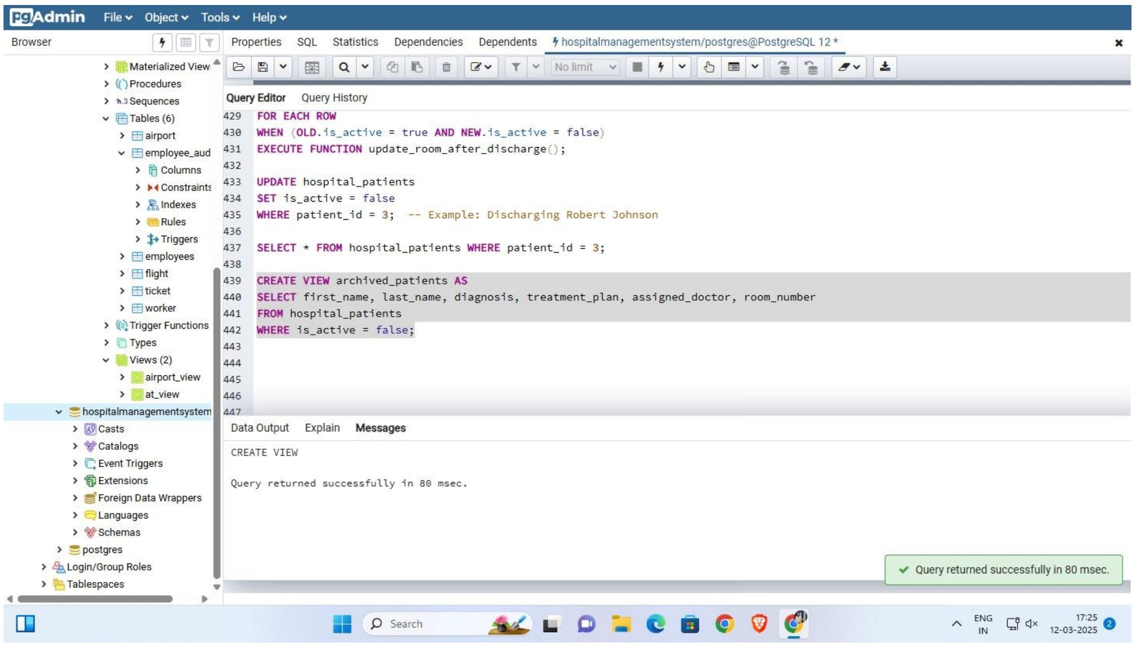
```
UPDATE 1

Query returned successfully in 54 msec.
```



```
    WHERE patient_id = NEW.patient_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_patient_discharge
AFTER UPDATE ON hospital_patients
FOR EACH ROW
WHEN (OLD.is_active = true AND NEW.is_active = false)
EXECUTE FUNCTION update_room_after_discharge();

UPDATE hospital_patients
SET is_active = false
WHERE patient_id = 3;  -- Example: Discharging Robert Johnson

SELECT * FROM hospital_patients WHERE patient_id = 3;
```

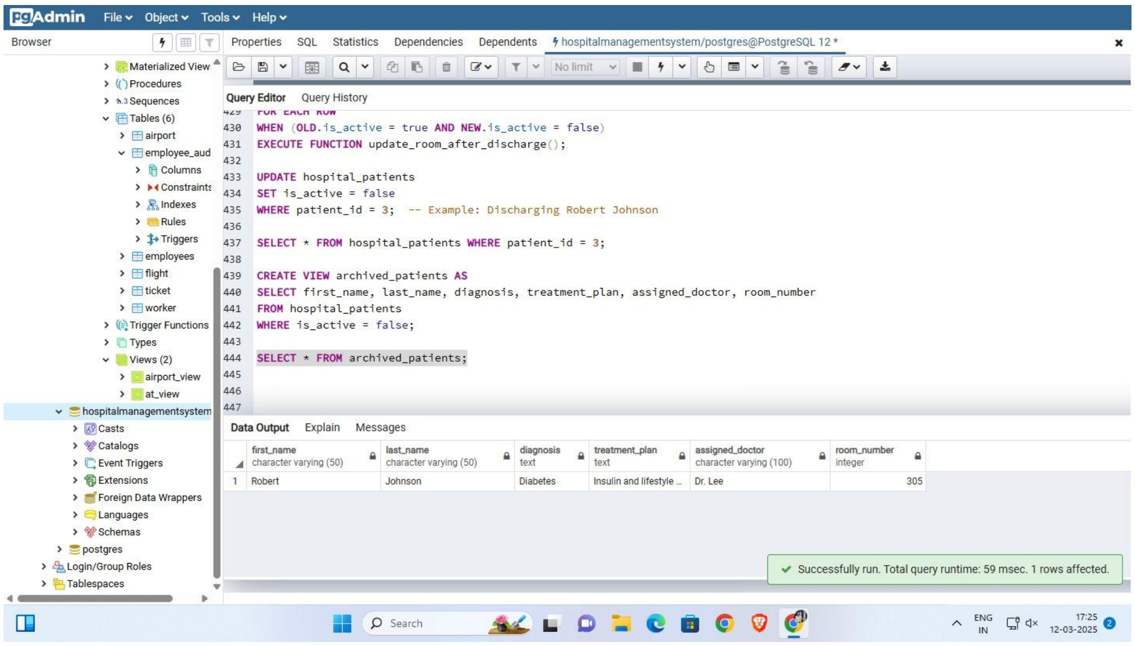
	patient_id [PK] integer	first_name character varying (50)	last_name character varying (50)	date_of_birth date	gender character varying (10)	admission_date date	discharge_date date	diagnosis text	treat text
1	3	Robert	Johnson	1992-09-09	Male	2024-01-15	2024-01-20	Diabetes	Insul

```
Successfully run. Total query runtime: 80 msec. 1 rows affected.
```



```
FOR EACH ROW
WHEN (OLD.is_active = true AND NEW.is_active = false)
EXECUTE FUNCTION update_room_after_discharge();

UPDATE hospital_patients
SET is_active = false
WHERE patient_id = 3;  -- Example: Discharging Robert Johnson

SELECT * FROM hospital_patients WHERE patient_id = 3;

CREATE VIEW archived_patients AS
SELECT first_name, last_name, diagnosis, treatment_plan, assigned_doctor, room_number
FROM hospital_patients
WHERE is_active = false;
```

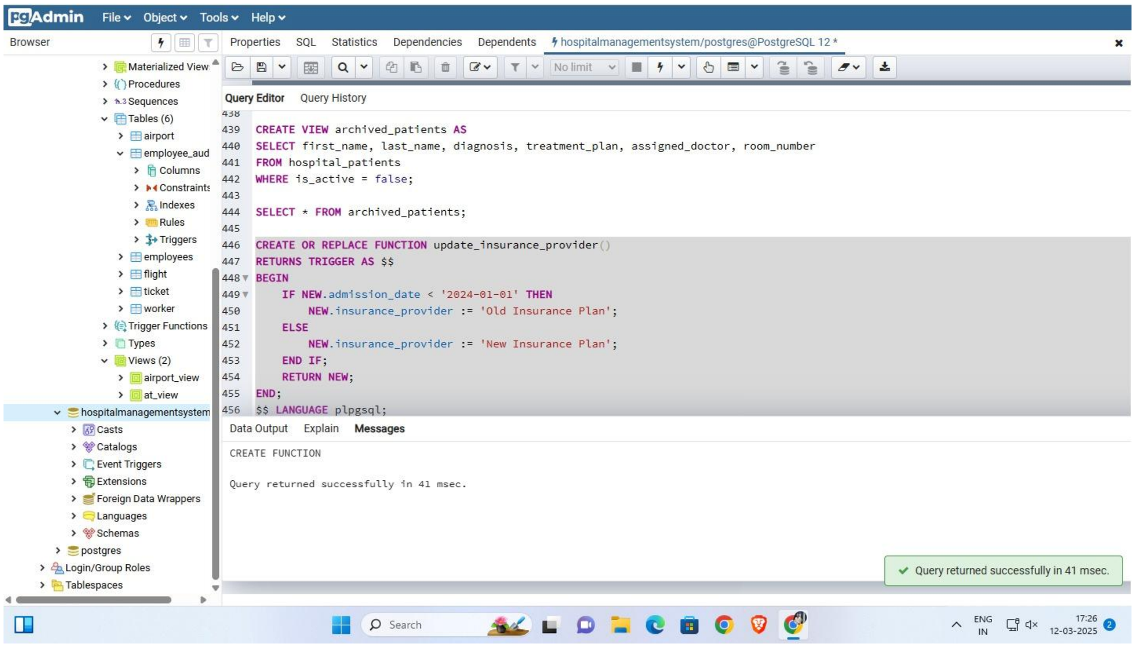
```
CREATE VIEW

Query returned successfully in 80 msec.
```



```
FOR EACH ROW
WHEN (OLD.is_active = true AND NEW.is_active = false)
EXECUTE FUNCTION update_room_after_discharge();

UPDATE hospital_patients
SET is_active = false
WHERE patient_id = 3;  -- Example: Discharging Robert Johnson

SELECT * FROM hospital_patients WHERE patient_id = 3;

CREATE VIEW archived_patients AS
SELECT first_name, last_name, diagnosis, treatment_plan, assigned_doctor, room_number
FROM hospital_patients
WHERE is_active = false;

SELECT * FROM archived_patients;
```

	first_name character varying (50)	last_name character varying (50)	diagnosis text	treatment_plan text	assigned_doctor character varying (100)	room_number integer
1	Robert	Johnson	Diabetes	Insulin and lifestyle ...	Dr. Lee	305

Successfully run. Total query runtime: 59 msec. 1 rows affected.

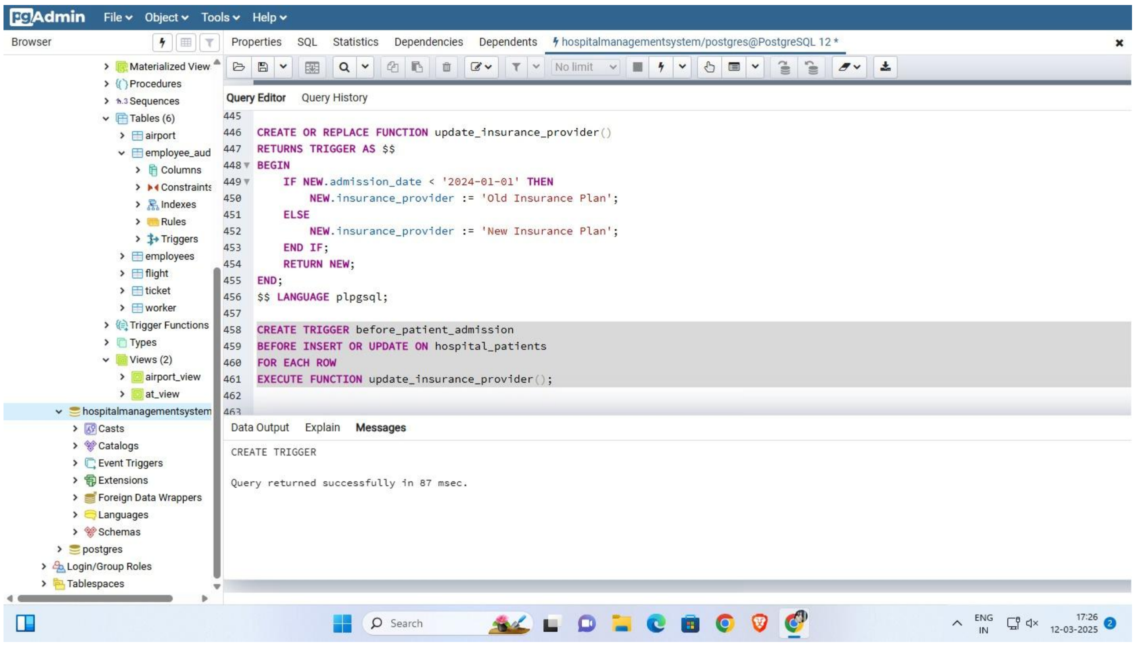


```
CREATE VIEW archived_patients AS
SELECT first_name, last_name, diagnosis, treatment_plan, assigned_doctor, room_number
FROM hospital_patients
WHERE is_active = false;

SELECT * FROM archived_patients;

CREATE OR REPLACE FUNCTION update_insurance_provider()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.admission_date < '2024-01-01' THEN
        NEW.insurance_provider := 'Old Insurance Plan';
    ELSE
        NEW.insurance_provider := 'New Insurance Plan';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION

Query returned successfully in 41 msec.
```

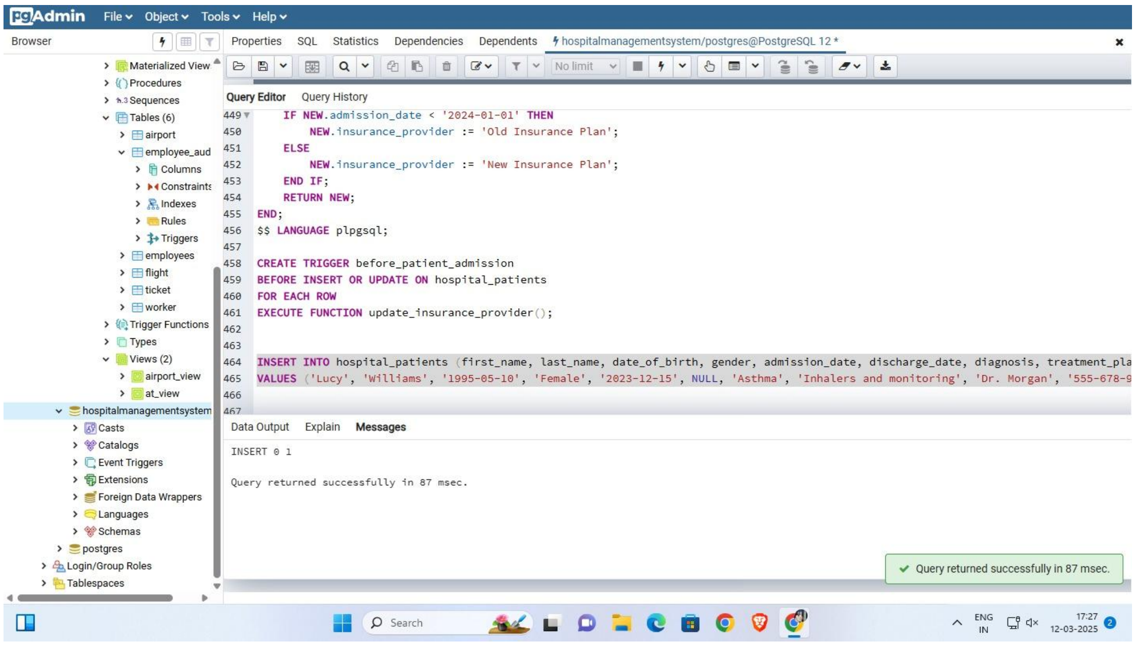
Query returned successfully in 41 msec.

TRIGGER



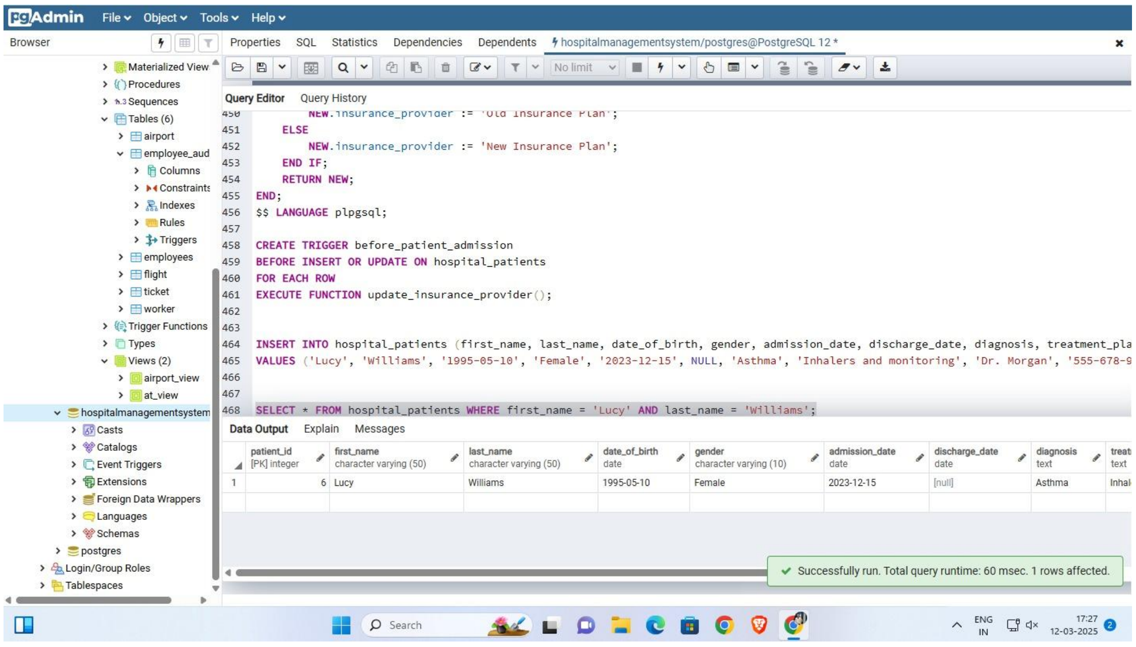
```
CREATE OR REPLACE FUNCTION update_insurance_provider()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.admission_date < '2024-01-01' THEN
        NEW.insurance_provider := 'Old Insurance Plan';
    ELSE
        NEW.insurance_provider := 'New Insurance Plan';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_patient_admission
BEFORE INSERT OR UPDATE ON hospital_patients
FOR EACH ROW
EXECUTE FUNCTION update_insurance_provider();
```

```
CREATE TRIGGER

Query returned successfully in 87 msec.
```



```
    IF NEW.admission_date < '2024-01-01' THEN
        NEW.insurance_provider := 'Old Insurance Plan';
    ELSE
        NEW.insurance_provider := 'New Insurance Plan';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_patient_admission
BEFORE INSERT OR UPDATE ON hospital_patients
FOR EACH ROW
EXECUTE FUNCTION update_insurance_provider();


INSERT INTO hospital_patients (first_name, last_name, date_of_birth, gender, admission_date, discharge_date, diagnosis, treatment_pla
VALUES ('Lucy', 'Williams', '1995-05-10', 'Female', '2023-12-15', NULL, 'Asthma', 'Inhalers and monitoring', 'Dr. Morgan', '555-678-9
```

```
INSERT 0 1

Query returned successfully in 87 msec.
```



```
            NEW.insurance_provider := 'Old Insurance Plan';
        ELSE
            NEW.insurance_provider := 'New Insurance Plan';
        END IF;
        RETURN NEW;
    END;
    $$ LANGUAGE plpgsql;

    CREATE TRIGGER before_patient_admission
    BEFORE INSERT OR UPDATE ON hospital_patients
    FOR EACH ROW
    EXECUTE FUNCTION update_insurance_provider();


    INSERT INTO hospital_patients (first_name, last_name, date_of_birth, gender, admission_date, discharge_date, diagnosis, treatment_pla
    VALUES ('Lucy', 'Williams', '1995-05-10', 'Female', '2023-12-15', NULL, 'Asthma', 'Inhalers and monitoring', 'Dr. Morgan', '555-678-9


    SELECT * FROM hospital_patients WHERE first_name = 'Lucy' AND last_name = 'Williams';
```

	patient_id [PK] integer	first_name character varying (50)	last_name character varying (50)	date_of_birth date	gender character varying (10)	admission_date date	discharge_date date	diagnosis text	treat text
1	6	Lucy	Williams	1995-05-10	Female	2023-12-15	[null]	Asthma	Inhal

```
Successfully run. Total query runtime: 60 msec. 1 rows affected.
```

Conclusion:

The above experiment highlights working with SQL queries based on views and triggers on our database.

Post Lab Questions:

1. 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 SQL query. Unlike a table, a view does not store data permanently; instead, it dynamically retrieves data from the underlying tables whenever accessed. Views are useful for simplifying complex queries, restricting access to certain columns, and improving security.

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

   ```
   CREATE VIEW EmployeeView AS

   SELECT Fname, Lname, Salary

   FROM employee;
   ```

3. Write a query to drop the trigger trg_city_update from the employee table.

   ```
   DROP TRIGGER trg_city_update ON employee;
   ```