Experiment - 7

Student Name: Anisha Kumari	**UID:** 23BAI70337
Branch: BE-AIT-CSE	**Section/Group:** 23AIT_KRG-G1_A
Semester: 5th	**Date of Performance:** 15 Oct, 2025
Subject Name: ADBMS	**Subject Code:** 23CSP-333

1. Aim:

MEDIUM LEVEL PROBLEM:
Design a PostgreSQL trigger such that whenever an insertion occurs on the *student* table, the currently inserted or deleted row details (ID, Name, Age, Class) should be printed exactly as they are in the output console.

HARD LEVEL PROBLEM:
Design a PostgreSQL trigger system where:

- When a new employee is inserted into *tbl_employee*, a record is added to *tbl_employee_audit* with the message:
  “Employee name <emp_name> has been added at <current_time>”.
- When an employee is deleted from *tbl_employee*, a record is added to *tbl_employee_audit* with the message:
  “Employee name <emp_name> has been deleted at <current_time>”.

2. Objective:

- To understand the concept and working of triggers in PostgreSQL.
- To learn how to create and implement trigger functions using PL/pgSQL.
- To utilize OLD and NEW records for handling row data before and after triggering events.
- To design automated auditing systems using triggers for data changes like INSERT and DELETE.

3. Theory:

A trigger in PostgreSQL is a special function that is automatically executed or fired in response to certain events on a table or view. These events can be **INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE** operations. Triggers help automate tasks, maintain data integrity, and perform complex business logic directly within the database.

In PostgreSQL, a trigger function is written in the **PL/pgSQL** language and is always associated with a table. The trigger function executes each time the specified event occurs on that table. There are two main types of triggers based on their timing: **BEFORE** triggers and **AFTER** triggers.

- A BEFORE trigger executes before the event occurs and can be used to validate or modify data.
- An AFTER trigger executes after the event has occurred and is often used for logging, auditing, or maintaining related tables.

Triggers can also be defined as **ROW-level** or **STATEMENT-level**. A ROW-level trigger fires once for each affected row, whereas a STATEMENT-level trigger executes once per SQL statement, regardless of how many rows are affected.

4. Procedure:

Medium Level Solution:

- Create or verify the student table with columns id (auto-generated), name, age, and class so the inserted row has an id available.
- Write a PL/pgSQL trigger function fn_student_audit() that checks TG_OP and uses RAISE NOTICE to print NEW.* on INSERT and OLD.* on DELETE, returning NEW for INSERT and OLD for DELETE.
- Create an AFTER trigger on student for INSERT OR DELETE, FOR EACH ROW, executing fn_student_audit() so a message is emitted for each affected row.
- Test by inserting and deleting a row from student; ensure client_min_messages allows NOTICE to display in your client.

Hard Level Solution:

- Create or verify the student table with columns id (auto-generated), name, age, and class so the inserted row has an id available.
- Write a PL/pgSQL trigger function fn_student_audit() that checks TG_OP and uses RAISE NOTICE to print NEW.* on INSERT and OLD.* on DELETE, returning NEW for INSERT and OLD for DELETE.
- Create an AFTER trigger on student for INSERT OR DELETE, FOR EACH ROW, executing fn_student_audit() so a message is emitted for each affected row.
- Test by inserting and deleting a row from student; ensure client_min_messages allows NOTICE to display in your client.

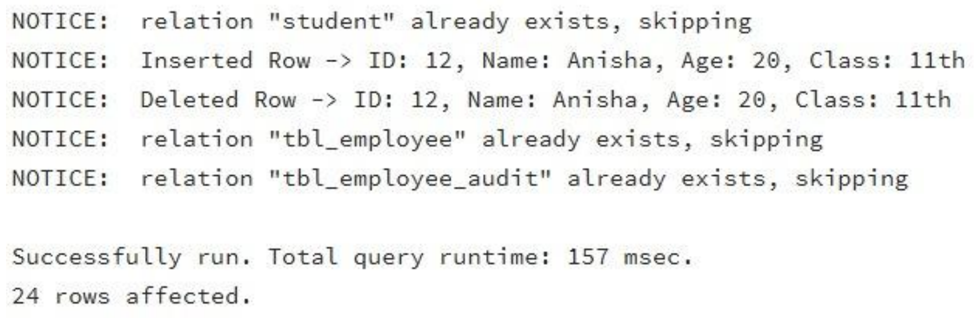
5. Code:

```
                    -----Experiment 7 (Medium Level Solution)-----

CREATE TABLE IF NOT EXISTS student (
    id SERIAL flRIMARY KEY,
    name VARCHAR(100),
    age INT,
    class VARCHAR(20)
);

CREATE OR REflLACE FUNCTION fn_student_audit()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF TG_Ofl = 'INSERT' THEN
        RAISE NOTICE 'Inserted Row -> ID: %, Name: %, Age: %, Class: %',
                     NEW.id, NEW.name, NEW.age, NEW.class;
        RETURN NEW;

    ELSIF TG_Ofl = 'DELETE' THEN
        RAISE NOTICE 'Deleted Row -> ID: %, Name: %, Age: %, Class: %',
                     OLD.id, OLD.name, OLD.age, OLD.class;
        RETURN OLD;
    END IF;

    RETURN NULL;
END;
$$;

DROfl TRIGGER IF EXISTS trg_student_audit ON student;

CREATE TRIGGER trg_student_audit
AFTER INSERT OR DELETE
ON student
FOR EACH ROW
EXECUTE    FUNCTION    fn_student_audit();

-- Insert
INSERT INTO student (name, age, class) VALUES ('Anisha', 19, '11th');


-- Delete
DELETE FROM student WHERE name = 'Anisha';
```

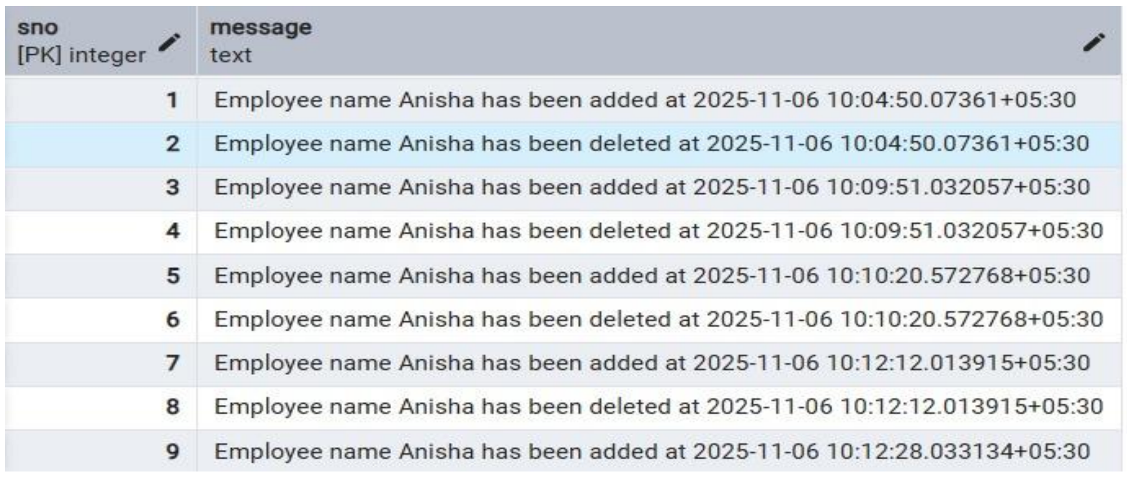
-----Experiment 7 (Hard Level Solution)-----

```
CREATE TABLE IF NOT EXISTS tbl_employee (
    emp_id SERIAL flRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    emp_salary NUMERIC
);

CREATE TABLE IF NOT EXISTS tbl_employee_audit (
    sno SERIAL flRIMARY KEY,
    message TEXT NOT NULL
);

CREATE OR REflLACE FUNCTION audit_employee_changes()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF TG_Ofl = 'INSERT' THEN
        INSERT INTO tbl_employee_audit(message)
        VALUES ('Employee name ' || NEW.emp_name || ' has been added at ' ||
NOW());
        RETURN NEW;

    ELSIF TG_Ofl = 'DELETE' THEN
        INSERT INTO tbl_employee_audit(message)
        VALUES ('Employee name ' || OLD.emp_name || ' has been deleted at ' ||
NOW());
        RETURN OLD;
    END IF;

    RETURN NULL;
END;
$$;

DROfl TRIGGER IF EXISTS trg_employee_audit ON tbl_employee;

CREATE TRIGGER trg_employee_audit
AFTER INSERT OR DELETE
ON tbl_employee
FOR EACH ROW
EXECUTE    FUNCTION    audit_employee_changes();

-- Insert and verify audit
INSERT INTO tbl_employee (emp_name, emp_salary) VALUES ('Anisha', 50000);
SELECT * FROM tbl_employee_audit;

-- Delete and verify audit
DELETE FROM tbl_employee WHERE emp_name = 'Anisha';
SELECT * FROM tbl_employee_audit;
```

6. Output:



```
NOTICE:  relation "student" already exists, skipping
NOTICE:  Inserted Row -> ID: 12, Name: Anisha, Age: 20, Class: 11th
NOTICE:  Deleted Row -> ID: 12, Name: Anisha, Age: 20, Class: 11th
NOTICE:  relation "tbl_employee" already exists, skipping
NOTICE:  relation "tbl_employee_audit" already exists, skipping

Successfully run. Total query runtime: 157 msec.
24 rows affected.
```



sno [PK] integer	message text
1	Employee name Anisha has been added at 2025-11-06 10:04:50.07361+05:30
2	Employee name Anisha has been deleted at 2025-11-06 10:04:50.07361+05:30
3	Employee name Anisha has been added at 2025-11-06 10:09:51.032057+05:30
4	Employee name Anisha has been deleted at 2025-11-06 10:09:51.032057+05:30
5	Employee name Anisha has been added at 2025-11-06 10:10:20.572768+05:30
6	Employee name Anisha has been deleted at 2025-11-06 10:10:20.572768+05:30
7	Employee name Anisha has been added at 2025-11-06 10:12:12.013915+05:30
8	Employee name Anisha has been deleted at 2025-11-06 10:12:12.013915+05:30
9	Employee name Anisha has been added at 2025-11-06 10:12:28.033134+05:30

7. Learning Outcomes:

- Explain what PostgreSQL triggers are, when they fire (INSERT, UPDATE, DELETE, TRUNCATE), and how timing types (BEFORE vs AFTER) affect behavior and use cases.
- Implement PL/pgSQL trigger functions that correctly use TG_OP along with NEW and OLD records to access row states for different events.
- Create and bind row-level triggers to tables using CREATE TRIGGER with the proper timing, events, and FOR EACH ROW vs FOR EACH STATEMENT semantics.
- Build practical auditing solutions: emit RAISE NOTICE messages showing affected row data, and persist human-readable audit logs to an audit table on INSERT and DELETE.
- Enforce business rules with BEFORE triggers to validate or block operations, and use RETURN NEW/OLD or NULL appropriately to allow or prevent changes.
- Test and troubleshoot triggers end-to-end, including verifying NOTICE visibility and confirming audit entries after DML operations.