Branch: MCA (Data Science) Kargil	**Semester:** 2
Student Name: Sameer Shahi	**UID:** 25MCD10025
Subject Name: Technical Training - I Lab	**Subject Code**: 25CAP-652
Section/Group: 25MCD KAR-1	**Date of Performance:** 27-02-2026

Experiment 5

Experiment Aim: To gain hands-on experience in creating and using cursors for row-by-row processing in a database, enabling sequential access and manipulation of query results for complex business logic. **(Company Tags: Infosys, Wipro, TCS, Capgemini)**

Tools used: PostgreSQL

Objectives

- **Sequential Data Access:** To understand how to fetch rows one by one from a result set using cursor mechanisms.
- **Row-Level Manipulation:** To perform specific operations or calculations on individual records that require conditional procedural logic.
- **Resource Management:** To learn the lifecycle of a cursor: Declaring, Opening, Fetching, and importantly, Closing and Deallocating to manage system memory.
- **Exception Handling:** To handle cursor-related errors and performance considerations during large-scale data iteration.

Theory: While SQL is generally set-oriented, certain tasks require a procedural approach where we process one row at a time. This is where **Cursors** are used:

1. **Cursor Types:** Cursors can be Implicit (managed by the system) or Explicit (defined by the developer). They can also be Forward-Only (moving only toward the end) or Scrollable (moving back and forth).
2. **The Lifecycle:** * **DECLARE:** Defines the SQL query for the cursor.

- **OPEN:** Executes the query and establishes the result set.
- **FETCH:** Retrieves a specific row into variables for processing.
- **CLOSE:** Releases the current result set.
- **DEALLOCATE:** Removes the cursor definition from memory.

3. **Use Case:** Cursors are ideal for generating row-specific reports, updating balances based on complex historical data, or migrating data where each record needs individual validation.

Experiment Steps:

Step 1: Implementing a Simple Forward-Only Cursor

Creating a cursor to loop through an Employee table and print individual records.

Query:

```
-- Query 1
-- Create Employee table (if not already created)
CREATE TABLE Employee (
   EmpID INT PRIMARY KEY,
   EmpName VARCHAR(50),
   Salary INT
);
-- Insert sample data
INSERT INTO Employee VALUES (1, 'Rahul', 30000);
INSERT INTO Employee VALUES (2, 'Priya', 35000);
INSERT INTO Employee VALUES (3, 'Amit', 40000);
-- Cursor declaration
DECLARE
   v_id Employee.EmpID%TYPE;
   v_name Employee.EmpName%TYPE;
   v_salary Employee.Salary%TYPE;
```

```
CURSOR emp_cursor IS
   SELECT EmpID, EmpName, Salary FROM Employee;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO v_id, v_name, v_salary;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(
         'ID: ' || v_id ||
         ', Name: ' || v_name ||
         ', Salary: ' || v_salary
      );
   END LOOP;
   CLOSE emp_cursor;
END;
/
```

Output:

Step 2: Complex Row-by-Row Manipulation

Using a cursor to update salaries based on a dynamic "Experience-to-Performance" ratio logic.

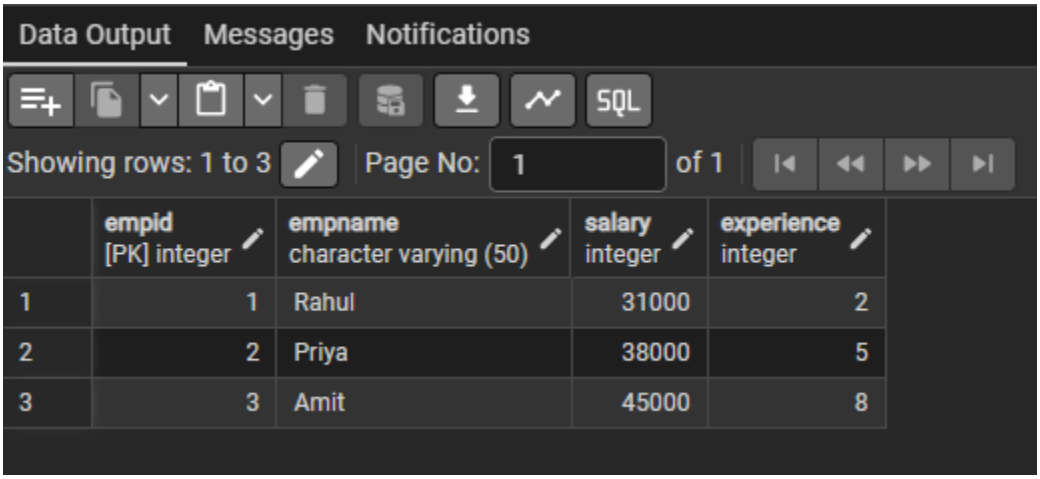
Query:

```
-- Query 2

-- Add Experience column

ALTER TABLE Employee

ADD COLUMN IF NOT EXISTS Experience INT;

-- Update sample experience values

UPDATE Employee SET Experience = 2 WHERE EmpID = 1;

UPDATE Employee SET Experience = 5 WHERE EmpID = 2;

UPDATE Employee SET Experience = 8 WHERE EmpID = 3;

-- Cursor to update salary based on experience

DO $$

DECLARE

    emp_record RECORD;

    emp_cursor CURSOR FOR

        SELECT EmpID, Salary, Experience FROM Employee;

BEGIN

    OPEN emp_cursor;

    LOOP

        FETCH emp_cursor INTO emp_record;
```

```
        EXIT WHEN NOT FOUND;

        -- Salary update logic

        IF emp_record.Experience >= 7 THEN

            UPDATE Employee

            SET Salary = Salary + 5000

            WHERE EmpID = emp_record.EmpID;

        ELSIF emp_record.Experience >= 4 THEN

            UPDATE Employee

            SET Salary = Salary + 3000

            WHERE EmpID = emp_record.EmpID;

        ELSE

            UPDATE Employee

            SET Salary = Salary + 1000

            WHERE EmpID = emp_record.EmpID;

        END IF;

    END LOOP;


    CLOSE emp_cursor;

END $$;


-- View updated table
```

SELECT * FROM Employee;

Output:



	empid [PK] integer	empname character varying (50)	salary integer	experience integer
1	1	Rahul	31000	2
2	2	Priya	38000	5
3	3	Amit	45000	8

Step 3: Exception and Status Handling

Ensuring the cursor handles empty result sets or termination signals gracefully.

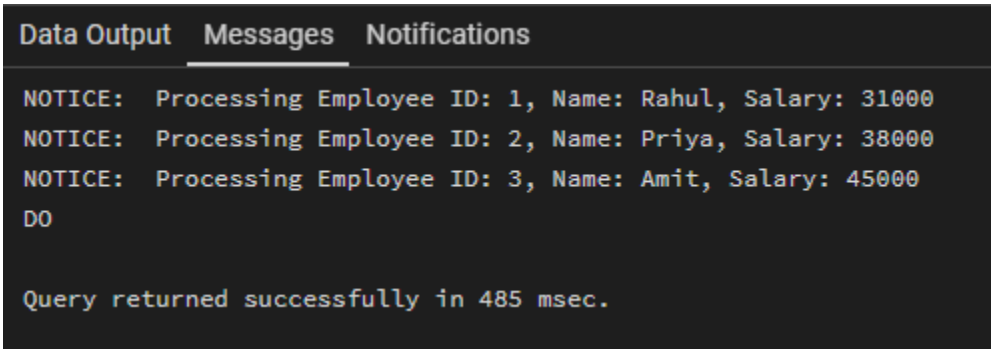
Query:

```
-- Query 3

DO $$

DECLARE

    emp_record RECORD;

    emp_cursor CURSOR FOR

        SELECT EmpID, EmpName, Salary FROM Employee;

BEGIN

    OPEN emp_cursor;
```

```
    -- Check if cursor has data
    FETCH emp_cursor INTO emp_record;

    IF NOT FOUND THEN
        RAISE NOTICE 'No records found in Employee table.';
    ELSE
        LOOP
            RAISE NOTICE 'Processing Employee ID: %, Name: %, Salary: %',
            emp_record.EmpID,
            emp_record.EmpName,
            emp_record.Salary;

            FETCH emp_cursor INTO emp_record;
            EXIT WHEN NOT FOUND;
        END LOOP;
    END IF;

    CLOSE emp_cursor;

EXCEPTION
    WHEN OTHERS THEN
```

```
    RAISE NOTICE 'An error occurred: %', SQLERRM;

END $$;
```

Output:



Outcomes:

- **Cursor Implementation:** Students will be able to design, implement, and manage cursors to solve row-wise processing problems.
- **Lifecycle Mastery:** Students will demonstrate the correct syntax for declaring, opening, fetching, and closing cursors.
- **Error Prevention:** Students will understand how to properly handle row-by-row processing exceptions and prevent memory leaks via deallocation.
- **Analytical Thinking:** Students will be able to apply cursor-based logic to solve real-world scenarios like multi-level payroll adjustments or data migrations.