Experiment 5

Student Name: Yash Karde **UID: 25MCI10090**
Branch: MCA (AI & ML) **Section/Group: 25MAM-1**
Semester: 2nd **Date of Performance:27/02/26**
Subject Name: Technical Training **Subject Code:25CAP-652**

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)

OBJECTIVE:

- **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.**

Implementation:

Step 1: Implementing a Simple Forward-Only Cursor

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

```
-- Table Creation

CREATE TABLE Employee (
    emp_id SERIAL PRIMARY KEY,
    emp_name VARCHAR(100),
    department VARCHAR(50),
    salary NUMERIC(10,2)
);
```

```
-- Data Insertion

INSERT INTO Employee (emp_name, department, salary) VALUES
('Rahul', 'IT', 50000),
('Sneha', 'HR', 40000),
('Amit', 'Finance', 55000),
('Priya', 'IT', 60000);

-- Cursor Commands

CREATE OR REPLACE FUNCTION display_employees()
RETURNS VOID AS
$$
DECLARE
   emp_record RECORD;
   emp_cursor CURSOR FOR SELECT * FROM Employee;
BEGIN
   OPEN emp_cursor;

   LOOP
      FETCH emp_cursor INTO emp_record;
      EXIT WHEN NOT FOUND;

      RAISE NOTICE 'ID: %, Name: %, Department: %, Salary: %',
         emp_record.emp_id,
         emp_record.emp_name,
         emp_record.department,
         emp_record.salary;
   END LOOP;

   CLOSE emp_cursor;
END;
$$
LANGUAGE plpgsql;

SELECT display_employees();
```

Output :-

```
NOTICE:  ID: 1, Name: Rahul, Department: IT, Salary: 50000.00
NOTICE:  ID: 2, Name: Sneha, Department: HR, Salary: 40000.00
NOTICE:  ID: 3, Name: Amit, Department: Finance, Salary: 55000.00
NOTICE:  ID: 4, Name: Priya, Department: IT, Salary: 60000.00

Successfully run. Total query runtime: 103 msec.
1 rows affected.
```

Step 2: Complex Row-by-Row Manipulation : -

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

```
ALTER TABLE Employee
ADD COLUMN experience INT,
ADD COLUMN performance_score INT;

UPDATE Employee SET experience = 5, performance_score = 8 WHERE emp_id = 1;
UPDATE Employee SET experience = 3, performance_score = 6 WHERE emp_id = 2;
UPDATE Employee SET experience = 7, performance_score = 9 WHERE emp_id = 3;
UPDATE Employee SET experience = 2, performance_score = 5 WHERE emp_id = 4;



CREATE OR REPLACE FUNCTION update_salary_by_ratio()
RETURNS VOID AS
$$
DECLARE
   emp_rec RECORD;
   exp_perf_ratio NUMERIC;
BEGIN
   FOR emp_rec IN SELECT * FROM Employee
   LOOP
      -- Ratio calculate karo
      exp_perf_ratio := emp_rec.performance_score::NUMERIC / emp_rec.experience;
      IF exp_perf_ratio > 1.5 THEN
```

```
            UPDATE Employee
            SET salary = salary * 1.20
            WHERE emp_id = emp_rec.emp_id;

        ELSIF exp_perf_ratio BETWEEN 1 AND 1.5 THEN
            UPDATE Employee
            SET salary = salary * 1.10
            WHERE emp_id = emp_rec.emp_id;

        ELSE
            UPDATE Employee
            SET salary = salary * 1.05
            WHERE emp_id = emp_rec.emp_id;
        END IF;
        RAISE NOTICE 'Updated Employee ID: %, Ratio: %',
            emp_rec.emp_id, exp_perf_ratio;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```

Output :-

Data Output | Messages | Notifications

```
NOTICE:  Updated Employee ID: 1, Ratio: 1.6000000000000000
NOTICE:  Updated Employee ID: 2, Ratio: 2.0000000000000000
NOTICE:  Updated Employee ID: 3, Ratio: 1.2857142857142857
NOTICE:  Updated Employee ID: 4, Ratio: 2.5000000000000000

Successfully run. Total query runtime: 97 msec.
1 rows affected.
```

Data Output | Messages | Notifications

	emp_id [PK] integer	emp_name character varying (100)	salary numeric (10,2)
1	1	Rahul	60000.00
2	2	Sneha	48000.00
3	3	Amit	60500.00
4	4	Priya	72000.00

Step 3: Exception and Status Handling

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

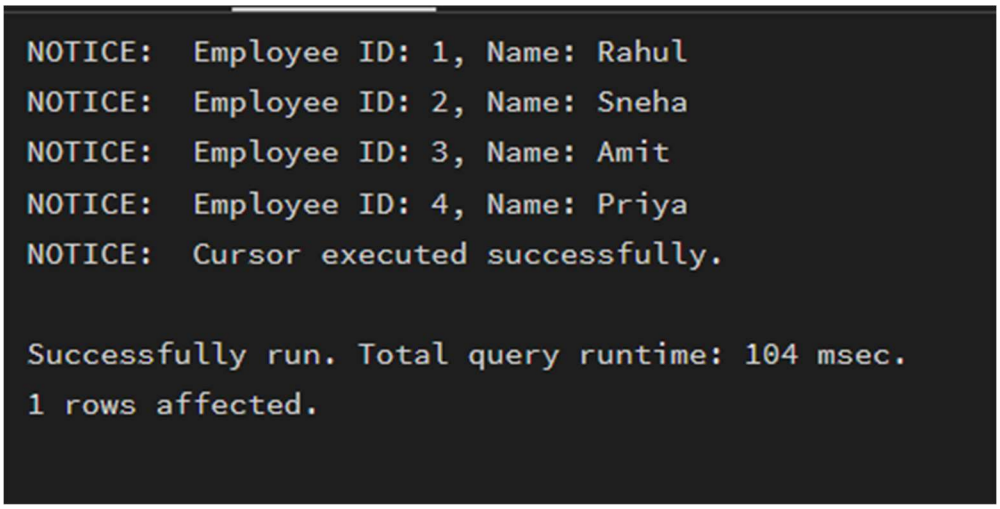
```
CREATE OR REPLACE FUNCTION safe_cursor_example()
RETURNS VOID AS
$$
DECLARE
   emp_rec RECORD;
   emp_cursor CURSOR FOR SELECT * FROM Employee;
BEGIN
   OPEN emp_cursor;

   LOOP
      FETCH emp_cursor INTO emp_rec;
      EXIT WHEN NOT FOUND;   -- Handles empty set & termination safely
      RAISE NOTICE 'Employee ID: %, Name: %',
         emp_rec.emp_id,
         emp_rec.emp_name;
   END LOOP;

   CLOSE emp_cursor;

   RAISE NOTICE 'Cursor executed successfully.';
END;
$$
LANGUAGE plpgsql;

SELECT safe_cursor_example();
```



LEARNING OUTCOMES:

1. **Cursor Implementation:** Students will be able to design, implement, and manage cursors to solve row-wise processing problems.

2. **Lifecycle Mastery:** Students will demonstrate the correct syntax for declaring, opening, fetching, and closing cursors.

3. **Error Prevention:** Students will understand how to properly handle row-by-row processing exceptions and prevent memory leaks via deallocation.

4. **Analytical Thinking:** Students will be able to apply cursor-based logic to solve real-world scenarios like multi-level payroll adjustments or data migrations.