Other Query Cluases and Lateral Joins

Advanced Query Techniques: Exercises

May 16, 2025

Contents

Global Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. Execute this script in your PostgreSQL environment before attempting the exercises.

```sql
-- SQL Dataset for PostgreSQL

-- Drop tables if they exist to ensure a clean slate
DROP TABLE IF EXISTS EmployeeProjects CASCADE;
DROP TABLE IF EXISTS ProductSales CASCADE;
DROP TABLE IF EXISTS Employees CASCADE;
DROP TABLE IF EXISTS Departments CASCADE;

-- Departments Table
CREATE TABLE Departments (
    departmentId SERIAL PRIMARY KEY,
    departmentName VARCHAR(100) UNIQUE NOT NULL,
    locationCity VARCHAR(50)
);

-- Employees Table
CREATE TABLE Employees (
    employeeId SERIAL PRIMARY KEY,
    firstName VARCHAR(50) NOT NULL,
    lastName VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    hireDate DATE NOT NULL,
    salary DECIMAL(10, 2) NOT NULL,
    departmentId INTEGER REFERENCES Departments(departmentId),
    managerId INTEGER -- Will add FK constraint later
);

-- ProductSales Table
CREATE TABLE ProductSales (
    saleId SERIAL PRIMARY KEY,
    productName VARCHAR(100) NOT NULL,
    category VARCHAR(50),
    saleDate TIMESTAMP NOT NULL,
    quantitySold INTEGER NOT NULL,
    unitPrice DECIMAL(10, 2) NOT NULL,
    region VARCHAR(50)
);

-- EmployeeProjects Table
CREATE TABLE EmployeeProjects (
    assignmentId SERIAL PRIMARY KEY,
    employeeId INTEGER NOT NULL, -- Will add FK constraint later
    projectName VARCHAR(100) NOT NULL,
    hoursWorked INTEGER,
    assignmentDate DATE
);

-- Populate Departments
INSERT INTO Departments (departmentName, locationCity) VALUES
('Human Resources', 'New York'),    -- departmentId 1
('Engineering', 'San Francisco'), -- departmentId 2
('Marketing', 'Chicago'),         -- departmentId 3
('Sales', 'Boston'),              -- departmentId 4
('Research', 'Austin');           -- departmentId 5

-- Populate Employees
-- Manually assigning employeeId for clarity in problem setup, SERIAL will handle it.
-- For inserts, rely on SERIAL. For managerId, use the IDs that will be generated.
-- Managers (NULL managerId or managerId referencing an already inserted employee)
INSERT INTO Employees (firstName, lastName, email, hireDate, salary, departmentId,
    managerId) VALUES
('Alice', 'Smith', 'alice.smith@example.com', '2020-01-15', 70000.00, 2, NULL),       --
    employeeId 1
('Diana', 'Prince', 'diana.prince@example.com', '2018-05-10', 150000.00, 1, NULL),    --
    employeeId 2
('Frank', 'Castle', 'frank.castle@example.com', '2017-11-05', 110000.00, 3, NULL),    --
    employeeId 3
```

```
('Henry', 'Jekyll', 'henry.jekyll@example.com', '2021-06-30', 88000.00, 4, NULL),  -- employeeId 4
('Kara', 'Stark', 'kara.stark@example.com', '2018-07-15', 130000.00, 5, NULL);  -- employeeId 5

-- Subordinate employees (managerId refers to employeeId generated above)
INSERT INTO Employees (firstName, lastName, email, hireDate, salary, departmentId, managerId) VALUES
('Bob', 'Johnson', 'bob.johnson@example.com', '2019-03-01', 120000.00, 2, 1),  -- employeeId 6
('Charlie', 'Brown', 'charlie.brown@example.com', '2021-07-22', 65000.00, 2, 1),  -- employeeId 7
('Eve', 'Adams', 'eve.adams@example.com', '2022-02-11', 50000.00, 1, 2),  -- employeeId 8
('Grace', 'Hopper', 'grace.hopper@example.com', '2020-08-19', 95000.00, 3, 3),  -- employeeId 9
('Ivy', 'Poison', 'ivy.poison@example.com', '2019-09-14', 72000.00, 4, 4),  -- employeeId 10
('Jack', 'Ripper', 'jack.ripper@example.com', '2022-01-01', 60000.00, 4, 4),  -- employeeId 11
('Leo', 'Martin', 'leo.martin@example.com', '2023-01-20', 55000.00, 5, 5),  -- employeeId 12
('Mia', 'Wallace', 'mia.wallace@example.com', '2020-04-05', 90000.00, 2, 1),  -- employeeId 13
('Noah', 'Chen', 'noah.chen@example.com', '2021-11-12', 75000.00, 3, 3),  -- employeeId 14
('Olivia', 'Davis', 'olivia.davis@example.com', '2022-05-25', 62000.00, 1, 2);  -- employeeId 15

-- Add self-referencing foreign key for Employees.managerId
ALTER TABLE Employees ADD CONSTRAINT fkManager FOREIGN KEY (managerId) REFERENCES Employees(employeeId);

-- Add foreign key for EmployeeProjects.employeeId
ALTER TABLE EmployeeProjects ADD CONSTRAINT fkEmployeeProjectsEmployee FOREIGN KEY (employeeId) REFERENCES Employees(employeeId);


-- Populate ProductSales (20 rows)
INSERT INTO ProductSales (productName, category, saleDate, quantitySold, unitPrice, region) VALUES
('Laptop Pro', 'Electronics', '2023-01-10 10:00:00', 5, 1200.00, 'North'),
('Smartphone X', 'Electronics', '2023-01-12 11:30:00', 10, 800.00, 'North'),
('Office Chair', 'Furniture', '2023-01-15 14:20:00', 2, 150.00, 'West'),
('Desk Lamp', 'Furniture', '2023-01-18 09:00:00', 3, 40.00, 'West'),
('Laptop Pro', 'Electronics', '2023-02-05 16:00:00', 3, 1200.00, 'South'),
('Smartphone X', 'Electronics', '2023-02-08 10:10:00', 8, 810.00, 'East'),
('Coffee Maker', 'Appliances', '2023-02-12 13:00:00', 1, 70.00, 'North'),
('Blender', 'Appliances', '2023-02-15 15:45:00', 2, 50.00, 'South'),
('Laptop Pro', 'Electronics', '2023-03-01 12:00:00', 4, 1180.00, 'West'),
('Smartphone X', 'Electronics', '2023-03-04 17:00:00', 12, 790.00, 'North'),
('Office Chair', 'Furniture', '2023-03-07 11:00:00', 1, 155.00, 'East'),
('Desk Lamp', 'Furniture', '2023-03-10 09:30:00', 5, 38.00, 'South'),
('Toaster', 'Appliances', '2023-03-13 14:50:00', 2, 30.00, 'West'),
('Vacuum Cleaner', 'Appliances', '2023-03-16 18:00:00', 1, 200.00, 'North'),
('Gaming Mouse', 'Electronics', '2023-04-01 10:00:00', 20, 50.00, 'East'),
('Keyboard', 'Electronics', '2023-04-02 11:00:00', 15, 75.00, 'West'),
('Monitor', 'Electronics', '2023-04-03 12:00:00', 7, 300.00, 'South'),
('External HDD', 'Electronics', '2023-04-04 13:00:00', 10, 80.00, 'North'),
('Webcam', 'Electronics', '2023-04-05 14:00:00', 12, 60.00, 'East'),
('Printer', 'Electronics', '2023-04-06 15:00:00', 4, 150.00, 'West');

-- Populate EmployeeProjects (10 rows)
-- employeeId values correspond to the SERIAL generated IDs:
-- Alice=1, Bob=6, Charlie=7, Eve=8, Grace=9, Ivy=10, Leo=12, Mia=13.
INSERT INTO EmployeeProjects (employeeId, projectName, hoursWorked, assignmentDate) VALUES
(1, 'Alpha Platform', 120, '2023-01-01'),
(6, 'Alpha Platform', 150, '2023-01-01'),
(7, 'Beta Feature', 80, '2023-02-15'),
(1, 'Beta Feature', 60, '2023-02-15'),
(8, 'HR Portal Update', 100, '2023-03-01'),
(9, 'Marketing Campaign Q1', 160, '2023-01-10'),
```

```
(10, 'Sales Dashboard', 130, '2023-02-01'),
(6, 'Gamma Initiative', 200, '2023-04-01'),
(13, 'Gamma Initiative', 180, '2023-04-01'),
(12, 'Research Paper X', 90, '2023-03-20');
```

Listing 1: Global Dataset for Exercises

1 Category: Other Query Clauses (FETCH, OFFSET)

1.1 (i) Practice meanings, values, relations, unique usage, and advantages

1.1.1 Exercise 1: Meaning of OFFSET and FETCH

Problem: Retrieve the product sales from the 6th to the 10th most recent sale (inclusive). Display 'saleId', 'productName', and 'saleDate'.

1.1.2 Exercise 2: Unique usage of FETCH with OFFSET for specific slicing

Problem: List all employees, but skip the first 3 highest paid employees and then show the next 5 highest paid after those. Display 'employeeId', 'firstName', 'lastName', and 'salary'.

1.2 (ii) Practice disadvantages of all its technical concepts

1.2.1 Exercise 3: Disadvantage of OFFSET without ORDER BY

Problem: Show the second page of 5 product sales using `OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY` but *without* an `ORDER BY` clause. Run the query multiple times. Are the results always the same? Explain the disadvantage.

1.2.2 Exercise 4: Disadvantage of large OFFSET - Performance

Problem: Imagine the 'ProductSales' table has millions of rows. Explain the potential performance disadvantage of fetching a page deep into the result set (e.g., `OFFSET 1000000 ROWS FETCH NEXT 10 ROWS ONLY`) when ordered by 'saleDate'.

1.3 (iii) Practice cases where people use inefficient basic solutions instead

1.3.1 Exercise 5: Inefficient pagination attempts vs. OFFSET/FETCH

Problem: Display the 3rd "page" of employees (3 employees per page) when ordered by 'hireDate' (oldest first). A page means a set of 3 employees. The 3rd page would be employees 7, 8, and 9 in the ordered list.

a) Describe or attempt to implement a common, but potentially less direct or efficient, SQL-based way this might be solved if a developer is unaware of or avoids `OFFSET/FETCH`. Consider approaches like using only `LIMIT` and fetching more data than needed, or attempting to simulate row skipping through complex subqueries (without using window functions like `ROW_NUMBER()`).

b) Solve the same problem using `OFFSET` and `FETCH`.

c) Discuss why `OFFSET/FETCH` is generally preferred for pagination.

1.4 (iv) Practice a hardcore problem combining previous concepts

1.4.1 Exercise 6: Hardcore OFFSET/FETCH with joins, set operations, subqueries, and filtering

Problem:

1. Create a combined list of employees from two specific groups:
    - Group A: All employees from the 'Engineering' department whose salary is $70,000 or more.
    - Group B: All employees from the 'Marketing' department whose hire date is on or after '2020-01-01'.
2. From this combined list, remove any duplicates based on 'employeeId'.
3. Order the resulting unique employees by their 'lastName' alphabetically (A-Z), then by 'firstName' alphabetically (A-Z).
4. From this final ordered list, retrieve the employees from the 2nd to the 3rd position (inclusive).
5. Display the 'employeeId', full name (concatenated 'firstName' and 'lastName' with a space), 'departmentName', 'salary', and 'hireDate' for these selected employees.

2 Category: LATERAL Joins

2.1 (i) Practice meanings, values, relations, unique usage, and advantages

2.1.1 Exercise 1: Meaning and unique usage of LATERAL - Top N per group

Problem: For each department, list the top 2 employees with the highest salary. Use `LIMIT 2` within the `LATERAL` subquery. If there are ties in salary that would extend beyond the top 2 individuals, your query should strictly return only 2 employees per department based on the salary ordering (and any secondary ordering specified). Display 'departmentName', 'employeeId', 'firstName', 'lastName', and 'salary'.

2.1.2 Exercise 2: LATERAL with a function-like subquery producing multiple related rows

Problem: For each product sale in the 'Electronics' category made in '2023-03-01' or later, calculate its total revenue ('quantitySold * unitPrice'). Then, list up to 2 *other* sales for the *same product* that occurred *earlier* than the current sale, ordered by the earlier sale date descending (most recent of the earlier sales first). Display the 'saleId', 'productName', and calculated total revenue of the current 'Electronics' sale, and the 'saleId', 'saleDate', and 'quantitySold' of the (up to) two prior sales for that product.

2.2 (ii) Practice disadvantages of all its technical concepts

2.2.1 Exercise 3: Disadvantage of LATERAL - Potential Performance Impact

Problem: For every employee, list their 'employeeId', 'firstName', 'lastName', and then use a `LATERAL` subquery to find up to 3 other employees in the *same department* who were hired *before* them and have a *higher salary*. Display the 'firstName', 'lastName', 'hireDate', and 'salary' of these senior, higher-paid colleagues. Discuss the potential performance disadvantage of this `LATERAL` join, especially if the 'Employees' table is very large and not optimally indexed for the subquery's conditions.

2.2.2 Exercise 4: Disadvantage - Readability/Complexity for simple cases

Problem: Retrieve all employees and their corresponding department names.

a) Solve this using a simple `INNER JOIN`.

b) Solve this using a `LATERAL` join where the subquery fetches the department name for the current employee's 'departmentId'.

c) Explain why using `LATERAL` here is an overkill and a disadvantage in terms of readability and simplicity for this specific task.

2.3 (iii) Practice cases where people use inefficient basic solutions instead

2.3.1 Exercise 5: Inefficient Top-1 per group without LATERAL

Problem: For each distinct 'region' in 'ProductSales', find the single product sale that had the highest total revenue (defined as 'quantitySold * unitPrice'). Display the 'region', 'productName', 'saleDate', and this highest total revenue. If multiple sales in a region share the same highest revenue, pick the one with the latest 'saleDate'. If there's still a tie, pick any.

a) Describe or attempt to implement a common (potentially inefficient or more complex) SQL-based approach to solve this *without* using `LATERAL` or window functions. Consider methods involving multiple queries with application-level joining of results, or complex correlated subqueries in the `SELECT` or `WHERE` clauses.

b) Show how to solve this efficiently and clearly using a `LATERAL` join.

c) Discuss why `LATERAL` (or window functions, though not the focus here) is superior to more basic, fragmented approaches for this "top-1-per-group" problem.

2.4 (iv) Practice a hardcore problem combining previous concepts

2.4.1 Exercise 6: Hardcore LATERAL with complex correlation, aggregation, and filtering

Problem: For each employee who is a manager (i.e., 'employeeId' appears as 'managerId' for at least one other employee):

1. Identify the top 2 most recent project assignments from the 'EmployeeProjects' table for *each employee they directly manage*.

2. For these selected project assignments (up to 2 per managed employee), calculate a "complexityScore" which is 'hoursWorked * (YEAR(assignmentDate) - 2020)'. Only consider projects with 'hoursWorked $>$ 50'. If 'YEAR(assignmentDate) - 2020' is less than 1, use 1 for that part of the calculation to avoid zero or negative scores.

3. Then, for each manager, calculate the sum of these "complexityScores" from all considered projects of their direct reports.

4. Display the manager's 'employeeId', 'firstName', 'lastName', and this total "sumComplexityScore".

5. Only include managers whose total "sumComplexityScore" is greater than 100.

6. Order the final result by the "sumComplexityScore" in descending order.