Other Query Cluases and Lateral Joins

Advanced Query Techniques: Solutions

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
```

```sql
('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 Other Query Clauses (FETCH, OFFSET)

1.1 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`. This exercise demonstrates the basic meaning of `OFFSET` (to skip a certain number of rows) and `FETCH` (to retrieve a specific number of subsequent rows) used together for pagination. It relies on `ORDER BY` for a stable order, which is crucial for meaningful pagination. The advantage is clear, standard SQL syntax for selecting a "slice" of an ordered result set.

Solution:

```sql
SELECT
    saleId,
    productName,
    saleDate
FROM
    ProductSales
ORDER BY
    saleDate DESC
OFFSET 5 ROWS -- Skip the first 5 most recent sales
FETCH NEXT 5 ROWS ONLY; -- Retrieve the next 5 sales (6th to 10th)
```

Listing 2: Solution to Other Query Clauses - Exercise 1

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`. This exercise highlights the use of `OFFSET` and `FETCH` to select a specific segment of an ordered dataset, not necessarily starting from the beginning. The unique usage is its directness for this "slice in the middle" scenario. The advantage is providing a standardized way for such pagination logic.

Solution:

```sql
SELECT
    employeeId,
    firstName,
    lastName,
    salary
FROM
    Employees
ORDER BY
    salary DESC
OFFSET 3 ROWS -- Skip the top 3 highest paid
FETCH NEXT 5 ROWS ONLY; -- Show the next 5
```

Listing 3: Solution to Other Query Clauses - Exercise 2

1.2 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. The disadvantage is that without `ORDER BY`, the order of rows in a table is not guaranteed. Running the query multiple times might yield different sets of 5 rows, making `OFFSET` and `FETCH` unreliable for consistent pagination.

Solution:

```sql
-- Query to run multiple times
SELECT saleId, productName, saleDate FROM ProductSales
OFFSET 5 ROWS
FETCH NEXT 5 ROWS ONLY;
```

Listing 4: Solution to Other Query Clauses - Exercise 3

Explanation of Disadvantage: SQL tables are inherently unordered sets of rows. Without an `ORDER BY` clause, the database system is free to return rows in any order it deems efficient, which can change due to various factors (e.g., updates, internal storage changes, query plan). Using `OFFSET` implies a sequence, but if that sequence isn't explicitly defined by `ORDER BY`, the rows skipped and fetched can be arbitrary and inconsistent between query executions. This makes pagination unreliable.

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`. The disadvantage is that many database systems must still identify, and often sort or scan, all the rows that come before the desired page (the 1,000,000 rows in the `OFFSET` clause) even if they are ultimately discarded. This can be very inefficient for large offset values.

Solution: Explanation of Disadvantage: When using a large `OFFSET` value, such as `OFFSET 1000000 ROWS`, the database system typically needs to:

1. Determine the correct order of all rows based on the `ORDER BY` clause (e.g., `ORDER BY saleDate`). This might involve sorting a large dataset if no suitable index exists or if the index cannot be used directly to skip to the offset.

2. Iterate through the ordered rows and count them until `1000000` rows are skipped.

3. Then, retrieve the next `10` rows.

The work done to sort/scan and then discard the first million rows is substantial and scales with the offset value. This makes "deep pagination" (accessing pages far from the beginning) slow. Alternative pagination strategies (like keyset/seek-based pagination) are often preferred for very large datasets to avoid this performance hit.

1.3 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.

- Part A: Show a common, but potentially less direct or efficient, way this might be attempted if a developer is unaware of or avoids `OFFSET/FETCH`.
- Part B: Solve the same problem using `OFFSET` and `FETCH`.
- Discuss why `OFFSET/FETCH` is generally preferred for pagination.

Solution: Part A: (Illustrative less direct approach) One might try to fetch records up to the end of the desired page and then, in application code, discard the earlier ones. A common bad practice is fetching ALL rows sorted, then an application layer skips and takes. If window functions were known, `ROW_NUMBER()` might be used as an alternative to `OFFSET/FETCH`.

```sql
/*
-- This solution uses ROW_NUMBER(), which is technically after OFFSET/FETCH in the course.
-- This is to illustrate an alternative that developers might reach for.
SELECT employeeId, firstName, lastName, hireDate, salary
FROM (
    SELECT
        employeeId,
        firstName,
        lastName,
        hireDate,
        salary,
        ROW_NUMBER() OVER (ORDER BY hireDate ASC) as rn
    FROM Employees
) AS Sub
WHERE rn > 6 AND rn <= 9; -- (pageNumber - 1) * pageSize < rn <= pageNumber * pageSize
                          -- (3 - 1) * 3 = 6 ; 3 * 3 = 9
*/

-- Another inefficient approach: Fetching more data than needed
-- Fetch 9 rows (up to the end of page 3)
-- SELECT employeeId, firstName, lastName, hireDate, salary FROM Employees ORDER BY hireDate ASC LIMIT 9;
-- Then, in application code, discard the first 6 and keep the last 3.
-- This transfers unnecessary data (first 6 rows).
```

Listing 5: Illustrative Inefficient Alternative (conceptual, uses later concepts)

Part B: Solution using `OFFSET` and `FETCH`: To get the 3rd page of 3 employees (employees 7, 8, 9): `OFFSET` should be `(pageNumber - 1) * pageSize = (3 - 1) * 3 = 6`. `FETCH` count is `pageSize = 3`.

```sql
SELECT
    employeeId,
    firstName,
    lastName,
    hireDate,
    salary
FROM
    Employees
ORDER BY
```

```
    hireDate ASC
OFFSET 6 ROWS -- Skip the first 2 pages (2 * 3 = 6 employees)
FETCH NEXT 3 ROWS ONLY; -- Fetch the 3 employees for the 3rd page
```

Listing 6: Solution to Other Query Clauses - Exercise 5 (Part B)

Discussion: `OFFSET` and `FETCH` (or `LIMIT offset, count`) are standard SQL clauses specifically designed for pagination. Advantages over alternatives:

1. **Clarity and Intent**: The syntax directly expresses the intent of skipping and taking a number of rows.

2. **Efficiency (generally)**: Database systems are often optimized for these clauses, especially when combined with `ORDER BY` on an indexed column. While large offsets can be slow, these clauses are generally more efficient than fetching large amounts of data to the client for client-side pagination or using very complex subqueries to simulate row numbering without proper window functions.

3. **Standardization**: `FETCH` and `OFFSET` are part of the SQL standard, promoting portability (though `LIMIT` is also common, its syntax varies).

Using `OFFSET/FETCH` avoids transferring unnecessary data to the application layer and keeps the pagination logic within the database, which is typically more efficient at data manipulation.

1.4 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.

Solution:

```sql
WITH CombinedEmployees AS (
    -- Group A: Engineering employees with salary >= $70,000
    SELECT E.employeeId, E.firstName, E.lastName, D.departmentName, E.salary, E.hireDate
    FROM Employees E
    JOIN Departments D ON E.departmentId = D.departmentId
    WHERE D.departmentName = 'Engineering' AND E.salary >= 70000.00

    UNION -- UNION removes duplicates between the two sets

    -- Group B: Marketing employees hired on or after '2020-01-01'
    SELECT E.employeeId, E.firstName, E.lastName, D.departmentName, E.salary, E.hireDate
    FROM Employees E
    JOIN Departments D ON E.departmentId = D.departmentId
    WHERE D.departmentName = 'Marketing' AND E.hireDate >= '2020-01-01'
),
OrderedEmployees AS (
    SELECT
        employeeId,
        firstName,
        lastName,
        departmentName,
        salary,
        hireDate,
        CONCAT(firstName, ' ', lastName) AS fullName
    FROM CombinedEmployees
    ORDER BY
        lastName ASC,
        firstName ASC
)
SELECT
    employeeId,
    fullName,
    departmentName,
    salary,
    hireDate
FROM OrderedEmployees
OFFSET 1 ROW -- Skip the 1st employee
FETCH NEXT 2 ROWS ONLY; -- Retrieve the 2nd and 3rd employees
```

Listing 7: Solution to Other Query Clauses - Exercise 6 (CTE version)

Alternative solution without CTEs (using subqueries in FROM):

```sql
SELECT
    OE.employeeId,
    CONCAT(OE.firstName, ' ', OE.lastName) AS fullName,
    OE.departmentName,
    OE.salary,
    OE.hireDate
FROM (
    SELECT
        CE.employeeId,
        CE.firstName,
        CE.lastName,
        CE.departmentName,
        CE.salary,
        CE.hireDate
```

```sql
    FROM (
        -- Group A: Engineering employees with salary >= $70,000
        SELECT E.employeeId, E.firstName, E.lastName, D.departmentName, E.salary, E.hireDate
        FROM Employees E
        JOIN Departments D ON E.departmentId = D.departmentId
        WHERE D.departmentName = 'Engineering' AND E.salary >= 70000.00

        UNION -- UNION removes duplicates between the two sets

        -- Group B: Marketing employees hired on or after '2020-01-01'
        SELECT E.employeeId, E.firstName, E.lastName, D.departmentName, E.salary, E.hireDate
        FROM Employees E
        JOIN Departments D ON E.departmentId = D.departmentId
        WHERE D.departmentName = 'Marketing' AND E.hireDate >= '2020-01-01'
    ) AS CE -- CombinedEmployees
    ORDER BY
        CE.lastName ASC,
        CE.firstName ASC
) AS OE -- OrderedEmployees
OFFSET 1 ROW -- Skip the 1st employee
FETCH NEXT 2 ROWS ONLY; -- Retrieve the 2nd and 3rd employees
```

Listing 8: Solution to Other Query Clauses - Exercise 6 (Subquery version)

2 LATERAL Joins

2.1 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. Display `departmentName`, `employeeId`, `firstName`, `lastName`, and `salary`. For this exercise, use `LIMIT 2` to get strictly two rows if available. This demonstrates `LATERAL`'s ability to perform a correlated subquery that returns multiple rows (the top N employees) for each row of the outer query (each department). This is a classic Top-N-per-group problem. The advantage is solving this often complex problem concisely and efficiently within SQL.

Solution:

```sql
SELECT
    D.departmentName,
    TopEmps.employeeId,
    TopEmps.firstName,
    TopEmps.lastName,
    TopEmps.salary
FROM
    Departments D
CROSS JOIN LATERAL (
    SELECT
        E.employeeId,
        E.firstName,
        E.lastName,
        E.salary
    FROM
        Employees E
    WHERE
        E.departmentId = D.departmentId -- Correlation
    ORDER BY
        E.salary DESC
    LIMIT 2 -- Get top 2 employees by salary for the current department
) AS TopEmps
ORDER BY
    D.departmentName ASC,
    TopEmps.salary DESC;
```

Listing 9: Solution to LATERAL Joins - Exercise 1

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 on '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. This exercise shows `LATERAL` joining an outer row (a specific sale) to a set of rows (prior sales of the same product) generated by a correlated subquery.

Solution:

```sql
SELECT
    PS_current.saleId AS currentSaleId,
    PS_current.productName,
    (PS_current.quantitySold * PS_current.unitPrice) AS currentSaleRevenue,
    PriorSales.priorSaleId,
    PriorSales.priorSaleDate,
    PriorSales.priorQuantitySold
FROM
    ProductSales PS_current
LEFT JOIN LATERAL ( -- Use LEFT JOIN LATERAL in case there are no prior sales
    SELECT
        PS_prior.saleId AS priorSaleId,
        PS_prior.saleDate AS priorSaleDate,
        PS_prior.quantitySold AS priorQuantitySold
    FROM
        ProductSales PS_prior
    WHERE
        PS_prior.productName = PS_current.productName  -- Correlation: same product
        AND PS_prior.saleDate < PS_current.saleDate     -- Correlation: earlier sale
    ORDER BY
        PS_prior.saleDate DESC -- Most recent of the earlier sales first
    LIMIT 2
) AS PriorSales ON TRUE -- ON TRUE is implicit for CROSS JOIN LATERAL, required for LEFT JOIN LATERAL
WHERE
    PS_current.category = 'Electronics' AND PS_current.saleDate >= '2023-03-01'
ORDER BY
    PS_current.productName, PS_current.saleDate, PriorSales.priorSaleDate DESC;
```

Listing 10: Solution to LATERAL Joins - Exercise 2

2.2 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. The disadvantage is that the `LATERAL` subquery is executed for each row of the outer `Employees` table. If the subquery involves scans or complex operations on a large table, and this is repeated N times, the total execution time can be very high.

Solution:

```sql
SELECT
    E1.employeeId AS currentEmployeeId,
```

```sql
    E1.firstName AS currentFirstName,
    E1.lastName AS currentLastName,
    SeniorColleagues.firstName AS seniorColleagueFirstName,
    SeniorColleagues.lastName AS seniorColleagueLastName,
    SeniorColleagues.hireDate AS seniorColleagueHireDate,
    SeniorColleagues.salary AS seniorColleagueSalary
FROM
    Employees E1
LEFT JOIN LATERAL (
    SELECT
        E2.firstName,
        E2.lastName,
        E2.hireDate,
        E2.salary
    FROM
        Employees E2
    WHERE
        E2.departmentId = E1.departmentId     -- Correlated: same department
        AND E2.hireDate < E1.hireDate         -- Correlated: hired before
        AND E2.salary > E1.salary             -- Correlated: higher salary
        AND E2.employeeId != E1.employeeId    -- Not the same employee
    ORDER BY
        E2.salary DESC, E2.hireDate DESC
    LIMIT 3
) AS SeniorColleagues ON TRUE
ORDER BY E1.employeeId, SeniorColleagues.salary DESC;
```

Listing 11: Solution to LATERAL Joins - Exercise 3

Explanation of Disadvantage: The `LATERAL` subquery is executed once for each employee (`E1`) in the outer query. If the `Employees` table has `N` rows, the subquery runs `N` times. Inside the subquery, it filters by `departmentId`, `hireDate`, and `salary`, then orders and limits. If `N` is large and the subquery cannot use indexes efficiently for its `WHERE` clause conditions, each execution might require a table scan or a significant index scan. This leads to a multiplicative effect on execution time (roughly `N * cost_of_subquery`). Poor indexing on `departmentId`, `hireDate`, and `salary` would exacerbate this.

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

Problem: Retrieve all employees and their corresponding department names.

- Part A: Solve this using a simple `INNER JOIN`.
- Part B: Solve this using a `LATERAL` join where the subquery fetches the department name for the current employee's `departmentId`.
- Explain why using `LATERAL` here is an overkill and a disadvantage.

The disadvantage is that `LATERAL` introduces unnecessary complexity for a straightforward join that can be easily and more clearly expressed with standard join syntax.

Solution: Part A: Using `INNER JOIN`

```sql
SELECT
    E.firstName,
```

```sql
    E.lastName,
    D.departmentName
FROM
    Employees E
INNER JOIN
    Departments D ON E.departmentId = D.departmentId;
```

Listing 12: Solution to LATERAL Joins - Exercise 4 (Part A)

Part B: Using `LATERAL JOIN`

```sql
SELECT
    E.firstName,
    E.lastName,
    DeptInfo.departmentName
FROM
    Employees E
CROSS JOIN LATERAL ( -- Assuming every employee has a valid departmentId
    SELECT
        D.departmentName
    FROM
        Departments D
    WHERE
        D.departmentId = E.departmentId -- Correlation
) AS DeptInfo;
```

Listing 13: Solution to LATERAL Joins - Exercise 4 (Part B)

Explanation of Disadvantage: For a simple lookup like fetching a department name, a standard `INNER JOIN` is highly readable and conventional. The `LATERAL` join in Part B is more verbose and complex for this task. While `LATERAL` is powerful for scenarios where the right side of the join depends on the left side in ways standard joins can't easily express (like top-N per group), using it for a basic foreign key lookup is overkill. It reduces query readability and can make maintenance harder.

2.3 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.

- Part A: Describe a common (potentially inefficient or more complex) way someone might try to solve this *without* using `LATERAL` or window functions.
- Part B: Show how to solve this efficiently and clearly using a `LATERAL` join.
- Discuss why `LATERAL` is superior for this "top-1-per-group" problem.

Solution: Part A: Inefficient/More Complex Approaches without LATERAL/Window Functions:

1. **Multiple Queries + Application Logic**:

   - Query 1: `SELECT region, MAX(quantitySold * unitPrice) AS maxRevenue FROM ProductSales GROUP BY region;`
   - Application Logic: Loop through results. For each region/maxRevenue, run:
   - Query 2 (per region): `SELECT productName, saleDate, (quantitySold * unitPrice) AS revenue FROM ProductSales WHERE region = 'current_region' AND (quantitySold * unitPrice) = current_maxRevenue ORDER BY saleDate DESC LIMIT 1;`

   This is inefficient due to multiple database roundtrips.

2. **Complex Correlated Subquery in SELECT (limited and inefficient)**:

```sql
/*
SELECT
    DISTINCT PS.region,
    (SELECT PS_inner.productName FROM ProductSales PS_inner
     WHERE PS_inner.region = PS.region
     ORDER BY (PS_inner.quantitySold * PS_inner.unitPrice) DESC, PS_inner.saleDate
    DESC LIMIT 1) AS topProductName,
    (SELECT (PS_inner.quantitySold * PS_inner.unitPrice) FROM ProductSales PS_inner
     WHERE PS_inner.region = PS.region
     ORDER BY (PS_inner.quantitySold * PS_inner.unitPrice) DESC, PS_inner.saleDate
    DESC LIMIT 1) AS topRevenue
    -- ... and so on for saleDate
FROM ProductSales PS;
*/
```

Listing 14: Illustrative Inefficient Correlated Subquery Approach

   This is very inefficient as subqueries are re-evaluated multiple times.

Part B: Solution using `LATERAL JOIN`

```sql
SELECT
    R.region,
    TopSale.productName,
    TopSale.saleDate,
    TopSale.totalRevenue
FROM
    (SELECT DISTINCT region FROM ProductSales) AS R -- Get unique regions
CROSS JOIN LATERAL (
    SELECT
        PS.productName,
        PS.saleDate,
        (PS.quantitySold * PS.unitPrice) AS totalRevenue
    FROM
        ProductSales PS
    WHERE
        PS.region = R.region -- Correlation
    ORDER BY
        totalRevenue DESC, PS.saleDate DESC -- Order by revenue, then by date
    LIMIT 1 -- Get the top 1 sale for this region
) AS TopSale
ORDER BY R.region;
```

Listing 15: Solution to LATERAL Joins - Exercise 5 (Part B)

Discussion: `LATERAL` provides a clean, SQL-native way to express "for each X, find the top related Y". Advantages:

1. **Single Query**: Reduces database roundtrips and application logic complexity.
2. **Readability**: Clear intent for those familiar with `LATERAL`.
3. **Performance**: Generally more efficient than multiple queries or repetitive correlated subqueries.
4. **Maintainability**: Centralized logic in one SQL query.

Developers unfamiliar with `LATERAL` might resort to less efficient methods, leading to performance bottlenecks or harder-to-maintain code.

2.4 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.
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.

Solution:

```sql
WITH Managers AS ( -- Identify managers
    SELECT DISTINCT E.employeeId AS managerEmployeeId, E.firstName, E.lastName
    FROM Employees E
    WHERE EXISTS (SELECT 1 FROM Employees Esub WHERE Esub.managerId = E.employeeId)
),
ManagerReportProjectScores AS (
    SELECT
        M.managerEmployeeId,
        M.firstName AS managerFirstName,
        M.lastName AS managerLastName,
        -- For each direct report of this manager
```

```sql
        -- Get their top 2 recent projects and calculate score
        COALESCE(ReportProjects.complexityScore, 0) AS projectComplexityScore
    FROM
        Managers M
    JOIN Employees Report ON Report.managerId = M.managerEmployeeId -- Join to get direct reports
    LEFT JOIN LATERAL ( -- Use LEFT JOIN LATERAL in case a report has no qualifying projects
        SELECT
            EP.projectName,
            EP.hoursWorked,
            EP.assignmentDate,
            (EP.hoursWorked * GREATEST(1, (EXTRACT(YEAR FROM EP.assignmentDate) - 2020)))::DECIMAL AS complexityScore
        FROM
            EmployeeProjects EP
        WHERE
            EP.employeeId = Report.employeeId -- Correlated to the direct report
            AND EP.hoursWorked > 50
        ORDER BY
            EP.assignmentDate DESC
        LIMIT 2 -- Top 2 recent projects for this report
    ) AS ReportProjects ON TRUE
)
SELECT
    MRS.managerEmployeeId,
    MRS.managerFirstName,
    MRS.managerLastName,
    SUM(MRS.projectComplexityScore) AS totalSumComplexityScore
FROM
    ManagerReportProjectScores MRS
GROUP BY
    MRS.managerEmployeeId, MRS.managerFirstName, MRS.managerLastName
HAVING
    SUM(MRS.projectComplexityScore) > 100
ORDER BY
    totalSumComplexityScore DESC;
```

Listing 16: Solution to LATERAL Joins - Exercise 6 (CTE version)

Alternative without CTEs (more complex nesting):

```sql
SELECT
    MRS.managerEmployeeId,
    MRS.managerFirstName,
    MRS.managerLastName,
    SUM(MRS.projectComplexityScore) AS totalSumComplexityScore
FROM (
    SELECT
        M.managerEmployeeId,
        M.firstName AS managerFirstName,
        M.lastName AS managerLastName,
        COALESCE(ReportProjects.complexityScore, 0) AS projectComplexityScore
    FROM
        (SELECT DISTINCT E.employeeId AS managerEmployeeId, E.firstName, E.lastName
```

```sql
         FROM Employees E
         WHERE EXISTS (SELECT 1 FROM Employees Esub WHERE Esub.managerId = E.employeeId)
        ) M -- Managers
    JOIN Employees Report ON Report.managerId = M.managerEmployeeId -- Direct reports
    LEFT JOIN LATERAL (
        SELECT
            (EP.hoursWorked * GREATEST(1, (EXTRACT(YEAR FROM EP.assignmentDate) - 2020)))::DECIMAL AS complexityScore
        FROM
            EmployeeProjects EP
        WHERE
            EP.employeeId = Report.employeeId
            AND EP.hoursWorked > 50
        ORDER BY
            EP.assignmentDate DESC
        LIMIT 2
    ) AS ReportProjects ON TRUE
) AS MRS -- ManagerReportProjectScores
GROUP BY
    MRS.managerEmployeeId, MRS.managerFirstName, MRS.managerLastName
HAVING
    SUM(MRS.projectComplexityScore) > 100
ORDER BY
    totalSumComplexityScore DESC;
```

Listing 17: Solution to LATERAL Joins - Exercise 6 (Subquery version)