Advanced SQL Query Techniques

Other Query Clauses & LATERAL Joins

Sequential SQL

May 16, 2025

Let's unlock some SQL magic for smarter data quests!

Contents

1 Other Query Clauses: `OFFSET` and `FETCH` 3

1.1 What Are They? (Meanings & Values) 3

1.2 Relations: How They Play with Others 3

1.3 How to Use Them: Structures & Syntax 4

1.4 Why Use Them? (Advantages) 5

1.5 Watch Out! (Disadvantages) 6

2 LATERAL Joins: The Row-by-Row Sidekick 7

2.1 What Are They? (Meanings & Values) 7

2.2 Relations: How They Play with Others 7

2.3 How to Use Them: Structures & Syntax 9

2.4 Why Use Them? (Advantages) 10

2.5 Watch Out! (Disadvantages) 11

1 Other Query Clauses: `OFFSET` and `FETCH`

Pagination and precise result slicing are common needs. Standard SQL offers `OFFSET` and `FETCH` for this, helping you grab just the data slice you need, no more, no less!

1.1 What Are They? (Meanings & Values)

Core Concepts: OFFSET and FETCH

These clauses fine-tune which rows your query returns from an ordered set.

- `OFFSET` ***count*** `[ROW | ROWS]`:
  - **Meaning**: This clause tells the database to skip the first *count* rows that would normally be returned by the query (after `ORDER BY` has done its job). It's like saying, "Start reading from page *count*+1."
  - **Value**: The *count* is a non-negative integer. If *count* is 0, no rows are skipped.
- `FETCH {FIRST | NEXT} [`*count*`] {ROW | ROWS} ONLY`:
  - **Meaning**: This clause specifies that only the next *count* rows should be returned, after any `OFFSET` rows have been skipped. `FIRST` and `NEXT` are synonyms. If *count* is omitted, it defaults to 1. "Just give me this many records, please!"
  - **Value**: The *count* is a positive integer.

The **output** is a subset of rows from the potentially larger result set defined by the rest of the query.

1.2 Relations: How They Play with Others

`OFFSET` and `FETCH` don't live in a vacuum; they interact with clauses you already know.

Synergies and Dependencies

Within "Other Query Clauses":

- `OFFSET` and `FETCH` are like partners in pagination. You often use them together, but `FETCH FIRST N ROWS ONLY` can be used alone (like `LIMIT N`). Using `OFFSET` alone is less common but syntactically allowed; it implies you want all rows *after* the offset.

Relations with Previous SQL Concepts:

- `ORDER BY` **(Basic SQL)**: This is the most crucial relationship! `OFFSET` and `FETCH` operate on a result set whose order is **defined by** `ORDER BY`. Without `ORDER BY`, the notion of "first rows to skip" or "next rows to fetch" is ambiguous and can lead to unpredictable results. *Always sort your list before you slice it, or the pieces might surprise you!*

    - Concepts like DESC, ordering by multiple columns, NULLS FIRST/LAST from Advanced ORDER BY are fully compatible and often necessary for stable pagination.

- LIMIT **(Basic SQL)**: In PostgreSQL and MySQL, LIMIT N OFFSET M is a common, non-standard shorthand. The SQL standard OFFSET M ROWS FETCH NEXT N ROWS ONLY is equivalent. FETCH FIRST N ROWS ONLY is the standard form of LIMIT N.

- WHERE **Clause (Basic SQL)**: Filtering with WHERE (including advanced conditions like subqueries in WHERE, BETWEEN, IN, LIKE) happens *before* ORDER BY, and thus before OFFSET and FETCH are applied. The pagination clauses work on the already filtered and sorted rows.

- **Joins (**INNER, LEFT, **etc. from Intermediate & Complementary SQL)**: Joins are resolved first to produce a combined result set. OFFSET and FETCH can then be applied to this joined, ordered result set.

- **Subqueries in** FROM **clause (Complementary SQL)**: You can apply OFFSET and FETCH to the result of a subquery. This is useful for paginating complex, preprocessed data.

```sql
SELECT *
FROM (
    SELECT employeeId, salary, hireDate
    FROM Employees
    WHERE departmentId = 2 -- From Engineering
    ORDER BY hireDate DESC -- Most recently hired first
) AS RecentHires
OFFSET 5 ROWS
FETCH NEXT 10 ROWS ONLY; -- Get 6th to 15th recent hire
```

Listing 1: Paginating a subquery's result

- **Set Operations (**UNION, INTERSECT, EXCEPT **from Complementary SQL)**: OFFSET and FETCH can be applied to the final result of a set operation, provided an ORDER BY clause is also applied to the overall result of the set operation.

1.3 How to Use Them: Structures & Syntax

Syntax Blueprints: OFFSET and FETCH

These clauses are typically placed at the end of a SELECT statement, after the ORDER BY clause.

1. Using OFFSET **only**:

```sql
SELECT columnList FROM YourTable
ORDER BY sortColumn
OFFSET numberToSkip ROWS;
```

Note: Skips numberToSkip *rows and returns all subsequent rows. Less common on its own for strict pagination.*

2. Using FETCH **only**:

```sql
SELECT columnList FROM YourTable
ORDER BY sortColumn
FETCH FIRST numberOfRowsToGet ROWS ONLY;
```

Note: This is the SQL standard equivalent of `LIMIT numberOfRowsToGet`.

3. Using `OFFSET` and `FETCH` together (most common for pagination):

```sql
SELECT columnList FROM YourTable
ORDER BY sortColumn
OFFSET numberToSkip ROWS
FETCH NEXT numberOfRowsToGet ROWS ONLY;
```

Example: To get the 3rd page of 10 items per page, you'd use `OFFSET 20 ROWS FETCH NEXT 10 ROWS ONLY`.

Keywords:

- `ROW` and `ROWS` are interchangeable.
- `FIRST` and `NEXT` are interchangeable in the `FETCH` clause.

PostgreSQL Alternative (`LIMIT` / `OFFSET`): Many databases, including PostgreSQL, offer a more concise (but non-standard) syntax:

```sql
-- Equivalent to FETCH FIRST N ROWS ONLY:
SELECT columnList FROM YourTable ORDER BY sortColumn LIMIT N;

-- Equivalent to OFFSET M ROWS FETCH NEXT N ROWS ONLY:
SELECT columnList FROM YourTable ORDER BY sortColumn LIMIT N
   OFFSET M;
```

While convenient in PostgreSQL, pgAdmin4 also fully supports the standard `OFFSET/FETCH` syntax. For portability, the standard syntax is preferred.

Example: Finding the 2nd and 3rd most expensive products

Let's say we have a `ProductSales` table.

```sql
SELECT productName, unitPrice, saleDate
FROM ProductSales
ORDER BY unitPrice DESC, saleDate DESC -- Primary sort by price, secondary by date
OFFSET 1 ROW -- Skip the most expensive one
FETCH NEXT 2 ROWS ONLY; -- Get the next two (2nd and 3rd)
```

Listing 2: Fetching 2nd and 3rd priciest sales

This query skips the single highest-priced sale and then retrieves the next two.

1.4 Why Use Them? (Advantages)

The Upsides of Slicing

- **Efficient Pagination**: This is their superpower! They allow you to break down vast result sets into smaller, digestible "pages" for user interfaces or batch processing. *No need to fetch the whole phonebook just to find one number!*

- **SQL Standard Compliance**: `OFFSET` and `FETCH` are part of the SQL standard (SQL:2008 and later), making your queries more portable across different database systems that adhere to the standard.

- **Clarity of Intent**: The syntax clearly expresses the desire to skip a certain number of rows and then take a specific number.

- **Top-N / Bottom-N Queries**: Easily retrieve the top or bottom N records according to some criteria when combined with `ORDER BY ... ASC/DESC` and `FETCH FIRST N ROWS ONLY`.

- **Resource Management**: By fetching only necessary data, you reduce data transfer over the network and memory consumption on the client side.

1.5 Watch Out! (Disadvantages)

Potential Pitfalls

- **Performance with Large `OFFSET` Values**: This is the big one. When you use a very large `OFFSET` (e.g., ‘OFFSET 1000000 ROWS‘), many database systems still need to internally generate, sort, and then count through all those million rows before they can fetch the few you asked for. This can be very slow. *Skipping a million pages in a book still means you touched them all!*

  - For "deep pagination," alternative techniques like keyset pagination (or "seek method") are often more performant. This involves using `WHERE` conditions on indexed columns from the last row of the previous page (e.g., ‘WHERE (lastValue, id) > ('someVal', 123)‘). (Keyset pagination uses concepts you already know but is a pattern, not a specific clause).

- **Crucial Dependency on `ORDER BY`**: If you forget `ORDER BY`, or if your `ORDER BY` clause doesn't produce a unique, stable order, the rows skipped and fetched can be inconsistent across query executions. The database gives no guarantee on row order without `ORDER BY`. *A shifty list makes for shifty slices!*

- **Non-Standard `LIMIT` Clause Prevalence**: Many developers are accustomed to the non-standard `LIMIT offset, count` or `LIMIT count OFFSET offset` syntax (e.g., MySQL, PostgreSQL, SQLite). While `OFFSET/FETCH` is standard, awareness of `LIMIT` variations is useful when working with different systems or older codebases.

2 LATERAL Joins: The Row-by-Row Sidekick

LATERAL joins are a powerful feature in SQL that allow a subquery in the FROM clause to reference columns from preceding tables in the same FROM clause. This enables complex, per-row calculations or lookups that are otherwise difficult or less efficient to express. Think of it as running a mini-query for each row of your main table!

2.1 What Are They? (Meanings & Values)

Core Concept: LATERAL Joins

- **Meaning**: A LATERAL join allows a derived table (subquery) or a table function on the right side of the join to be evaluated *for each row* from the table expression on its left. Critically, the right-hand side expression can **correlate** with columns from the left-hand side row currently being processed. It's like saying, "For this specific student, go find their top 3 recent test scores."
- **Keywords**: The keyword LATERAL is used immediately before the subquery or table function in the FROM clause.
  - `FROM LeftTable LT, LATERAL (subquery referencing LT.column) AS Sub`
  - `FROM LeftTable LT [INNER | LEFT | CROSS] JOIN LATERAL (subquery referencing LT.column) AS Sub ON ...`
- **Value/Output**: The result is a joined table.
  - If used with INNER JOIN LATERAL or CROSS JOIN LATERAL, rows from the left table are included only if the lateral subquery produces at least one row for them.
  - If used with LEFT JOIN LATERAL ... ON TRUE (or a suitable condition), all rows from the left table are included. If the lateral subquery produces no rows for a given left table row, columns from the lateral subquery will be NULL for that row.

The power comes from the subquery being re-evaluated dynamically for each outer row, using values from that outer row.

2.2 Relations: How They Play with Others

LATERAL joins unlock new ways to combine and process data by building upon familiar SQL concepts.

Connections and Context

Within "LATERAL Joins" Concepts:

- The LATERAL keyword itself is the enabler.
- It's typically combined with standard join types like INNER JOIN LATERAL, LEFT

`JOIN LATERAL`, or simply a comma followed by `LATERAL` (which behaves like `CROSS JOIN LATERAL` or an implicit `INNER JOIN LATERAL` if the subquery returns no rows for an outer row).

Relations with Previous SQL Concepts:

- **Subqueries in `FROM` clause (Complementary SQL)**: A `LATERAL` subquery *is* a subquery in the `FROM` clause. The key difference is its ability to reference preceding table columns. Regular subqueries in `FROM` cannot do this; they are evaluated independently. *LATERAL gives your subquery eyes to see its neighbors!*
- **Correlated Subqueries (Complementary SQL)**: `LATERAL` brings the power of correlation, traditionally seen in scalar subqueries in the `SELECT` list or in `WHERE` clause predicates (like `EXISTS`), directly into the `FROM` clause. Unlike scalar correlated subqueries that must return a single value, a `LATERAL` subquery can return an entire set of rows (a table) for each outer row.
- **Joins (`INNER`, `LEFT`, `CROSS` - Intermediate & Complementary SQL)**: `LATERAL` modifies the behavior of these joins. For example, `LEFT JOIN LATERAL` ensures all left-side rows are kept, even if the correlated subquery finds nothing for them.
- `WHERE` **Clause (Basic SQL)**:
  - *Inside* the `LATERAL` subquery: This is where the correlation happens, filtering rows of the subquery based on values from the current outer row.
  - *Outside* the `LATERAL` join (in the main query's `WHERE` clause): Filters the combined result set produced by the `LATERAL` join.
- `ORDER BY` / `LIMIT` **(or `OFFSET`/`FETCH`) (Basic SQL & Other Query Clauses)**: These are extremely powerful *inside* the `LATERAL` subquery. This is the standard way to solve "Top-N-per-group" problems (e.g., "for each department, get its top 3 highest-paid employees").
- **Aggregate Functions (`SUM`, `AVG`, etc. - Intermediate SQL)**:
  - *Inside* the `LATERAL` subquery: Can compute aggregate values for each outer row (e.g., "for each product, calculate its total sales from another table").
  - *Outside*, in the main query: Can aggregate results from the `LATERAL` join.
- **Scalar Subqueries & Subqueries in `SELECT` (Complementary SQL)**: `LATERAL` joins can often replace multiple, less efficient scalar correlated subqueries in the `SELECT` list, especially when you need several related pieces of information for each outer row.

2.3 How to Use Them: Structures & Syntax

Syntax Blueprints: LATERAL Joins

The `LATERAL` keyword is placed in the `FROM` clause.

1. Implicit Cross/Inner Join with LATERAL (Comma Syntax):

```
FROM LeftTableA A,
     LATERAL (
         SELECT SubColumn1, SubColumn2
         FROM SomeOtherTable SOT
         WHERE SOT.linkingColumn = A.columnFromA -- Correlation!
         -- ... other conditions, ORDER BY, LIMIT/FETCH ...
     ) AS SubqueryAlias
```

If the subquery returns no rows for a row in LeftTableA, that row from LeftTableA will NOT appear in the final result.

2. Explicit `JOIN LATERAL` (e.g., `LEFT JOIN LATERAL`):

```
FROM LeftTableA A
LEFT JOIN LATERAL ( -- Could be INNER JOIN LATERAL, CROSS JOIN
   LATERAL
    SELECT SubColumn1, SubColumn2
    FROM SomeOtherTable SOT
    WHERE SOT.linkingColumn = A.columnFromA -- Correlation!
    -- ... other conditions, ORDER BY, LIMIT/FETCH ...
) AS SubqueryAlias ON TRUE -- ON TRUE is common for LEFT JOIN
   LATERAL
                           -- when all logic is in the subquery.
```

With LEFT JOIN LATERAL, if the subquery returns no rows for a row in LeftTableA, that row from LeftTableA IS STILL included, with NULL values for columns from SubqueryAlias.

Key Use Case: Top-N per Group This is a classic `LATERAL` problem. Example: Get the latest 2 projects for each employee.

```
SELECT
    E.employeeId,
    E.firstName,
    E.lastName,
    P.projectName,
    P.assignmentDate
FROM
    Employees E
LEFT JOIN LATERAL (
    SELECT
        EP.projectName,
        EP.assignmentDate
    FROM
        EmployeeProjects EP
    WHERE
        EP.employeeId = E.employeeId -- Correlate to the current
           employee
    ORDER BY
        EP.assignmentDate DESC
    FETCH FIRST 2 ROWS ONLY -- Get the 2 most recent
) AS P ON TRUE -- ON TRUE as we want all employees
ORDER BY
```

```
    E.employeeId, P.assignmentDate DESC;
```

Listing 3: Top 2 recent projects per employee

This query shows each employee, and next to them, their two most recent projects. If an employee has fewer than two, it shows what they have. If none, project details are NULL.

Database Support:

- **PostgreSQL**: Full support for `LATERAL`.
- **MySQL**: Supported since version 8.0.14.
- **Oracle**: Supports `LATERAL`.
- **SQL Server**: Uses a similar concept called `APPLY` (`CROSS APPLY` and `OUTER APPLY`). `CROSS APPLY` is like an inner lateral join, and `OUTER APPLY` is like a left lateral join.

When working in pgAdmin4, you can use the standard `LATERAL` syntax.

2.4 Why Use Them? (Advantages)

The Power of Per-Row Processing

- **Solves "Top-N-per-Group" Elegantly**: This is a common and often tricky problem (e.g., "find the 3 most recent orders for each customer"). `LATERAL` with `ORDER BY` and `LIMIT`/`FETCH` inside the subquery is the canonical SQL solution. *Like giving each group manager a tool to pick their own top performers!*
- **Increased Readability for Complex Correlations**: For certain problems, `LATERAL` can make the query logic clearer than alternative solutions involving complex correlated scalar subqueries or convoluted self-joins. It explicitly states "for each row X, compute Y".
- **Flexibility in Subqueries**: The lateral subquery can be arbitrarily complex, involving its own joins, aggregations, etc., all parameterized by the current outer row.
- **Potential for Better Performance (sometimes)**: Compared to multiple correlated scalar subqueries in the `SELECT` list (each hitting the database potentially), a single `LATERAL` join might be optimized better by the database engine, especially if the subquery can efficiently use indexes based on the correlation columns.
- **Joining with Table-Valued Functions (TVFs)**: `LATERAL` is essential for passing columns from an outer table as arguments to a TVF that returns a set of rows. (While specific Set Returning Functions like `generate_series` are later in the course, the general advantage of `LATERAL` with function-like constructs is significant).

2.5 Watch Out! (Disadvantages)

⚠ Points to Ponder

- **Performance Considerations**: The lateral subquery is executed *for each row* of the outer table expression. If the outer table is large and the lateral subquery is complex or cannot use indexes efficiently, the query can become very slow. *Asking every person in a stadium for their life story takes time!*

  - Always ensure that the correlated conditions inside the `LATERAL` subquery can leverage indexes on the tables used within the subquery. Analyze query plans (`EXPLAIN`) for performance-critical `LATERAL` joins.

- **Increased Complexity for Beginners**: The concept of a subquery being re-evaluated for each outer row can be less intuitive than simple joins for those new to the idea. *It's a step up from a simple handshake to a coordinated dance move.*

- **Overkill for Simple Cases**: If a standard `JOIN` (e.g., `INNER JOIN` on a foreign key) can achieve the same result, using `LATERAL` would be unnecessarily complex and verbose. Stick to the simplest effective tool.

- **Syntax Variations (e.g., `APPLY` in SQL Server)**: While `LATERAL` is standardizing, if you work across different RDBMS, you might encounter vendor-specific alternatives like `APPLY` in SQL Server, which function similarly but have different syntax.

With `OFFSET`, `FETCH`, and `LATERAL` in your SQL toolkit, you're now equipped to craft even more precise and powerful data queries. Happy querying!