Subqueries, CTEs, and Window Functions: Advanced Query Composition

Subqueries: Queries Within Queries

A subquery is a SELECT statement nested inside another query. They let you build complex logic in stages.

Types of Subqueries

1. **Scalar subqueries:** Return a single value
2. **Row subqueries:** Return a single row (uncommon)
3. **Table subqueries:** Return multiple rows/columns
4. **Correlated subqueries:** Reference outer query

Scalar Subqueries: Single Value

Use case: "For each user, show how many orders they have"

```
SELECT
  name,
  (SELECT COUNT(*) FROM orders WHERE user_id = users.id) AS order_count
FROM users;
```

What happens:

- For each user, the subquery runs
- It counts orders for that specific user
- Returns a single number

Performance note: The subquery runs **once per user**. If you have 10,000 users, it runs 10,000 times.

This is called a **correlated subquery** because it references `users.id` from the outer query.

Table Subqueries: Multiple Rows

Use case: "Show orders placed by users in New York"

```
SELECT *
FROM orders
WHERE user_id IN (
  SELECT id FROM users WHERE city = 'New York'
);
```

What happens:

- Inner query returns a set of user IDs
- Outer query filters orders against that set

This is uncorrelated: The subquery doesn't reference the outer query.

Subqueries in FROM: Derived Tables

You can treat a subquery as a table:

```sql
SELECT avg_price, COUNT(*) AS product_count
FROM (
  SELECT category_id, AVG(price) AS avg_price
  FROM products
  GROUP BY category_id
) AS category_avg
GROUP BY avg_price;
```

What happens:

1. Inner query computes average price per category
2. Outer query treats that result as a table
3. Groups by those averages

Note: You must alias subqueries in FROM (`AS category_avg`).

Correlated vs Uncorrelated Subqueries

Uncorrelated Subqueries

Independent of outer query:

```sql
SELECT name
FROM products
WHERE category_id IN (
  SELECT id FROM categories WHERE active = true
);
```

Execution:

1. Inner query runs once
2. Outer query uses the result

Performance: Usually fast (inner query runs once).

Correlated Subqueries

Depends on outer query:

```sql
SELECT name
FROM products p
WHERE price > (
  SELECT AVG(price) FROM products WHERE category_id = p.category_id
);
```

Execution:

1. For each product, inner query runs
2. Computes average for that product's category
3. Compares product price to that average

Performance: Can be slow (inner query runs once per outer row).

When the optimizer saves you: Sometimes the database optimizes correlated subqueries into joins. Sometimes it doesn't. Use EXPLAIN to check.

Converting Correlated to Uncorrelated

Correlated (slow):

```sql
SELECT
  u.name,
  (SELECT COUNT(*) FROM orders WHERE user_id = u.id) AS order_count
FROM users u;
```

Uncorrelated with JOIN (faster):

```sql
SELECT
  u.name,
  COUNT(o.id) AS order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.name;
```

Why it's faster: Single pass through both tables instead of N queries.

CTEs (Common Table Expressions): WITH Clause

CTEs let you name subqueries for readability and reuse.

Basic Syntax

```sql
WITH high_value_orders AS (
  SELECT * FROM orders WHERE total > 1000
)
SELECT user_id, COUNT(*) AS big_order_count
FROM high_value_orders
GROUP BY user_id;
```

Equivalent to:

```sql
SELECT user_id, COUNT(*) AS big_order_count
FROM (
  SELECT * FROM orders WHERE total > 1000
) AS high_value_orders
GROUP BY user_id;
```

Why use CTEs?

- **Readability:** Named stages are clearer than nested subqueries
- **Reusability:** Reference the CTE multiple times in the main query
- **Maintainability:** Easier to modify one piece

Multiple CTEs

```sql
WITH
  active_users AS (
    SELECT id, name FROM users WHERE active = true
  ),
  recent_orders AS (
    SELECT user_id, total FROM orders WHERE created_at > NOW() - INTERVAL '30 days'
  )
SELECT
  u.name,
  SUM(o.total) AS recent_revenue
FROM active_users u
JOIN recent_orders o ON u.id = o.user_id
GROUP BY u.id, u.name;
```

Each CTE is defined once, used as needed.

Recursive CTEs: For Hierarchical Data

Use case: Org chart (employee → manager hierarchy).

```sql
WITH RECURSIVE employee_hierarchy AS (
  -- Base case: top-level employees (no manager)
  SELECT id, name, manager_id, 1 AS level
  FROM employees
  WHERE manager_id IS NULL

  UNION ALL

  -- Recursive case: employees with managers
  SELECT e.id, e.name, e.manager_id, eh.level + 1
  FROM employees e
  JOIN employee_hierarchy eh ON e.manager_id = eh.id
)
SELECT * FROM employee_hierarchy
ORDER BY level, name;
```

How it works:

1. Base case seeds the recursion (top-level managers)
2. Recursive case joins employees to previous level
3. Continues until no more rows are added
4. Returns the full hierarchy

Common use cases:

- Org charts
- Bill of materials (parts and subparts)
- File systems, category trees
- Graph traversal

Warning: Recursive CTEs can infinite loop if your data has cycles. Add a depth limit:

```
WHERE eh.level < 10
```

The CTE "Optimization Fence" Myth

There's a widespread belief that CTEs in PostgreSQL create an "optimization fence" that prevents the optimizer from inlining them.

The Nuance

Before Postgres 12: CTEs were always materialized (computed once, stored temporarily). The optimizer couldn't push predicates into them.

Since Postgres 12: CTEs are inlined by default (treated like subqueries). The optimizer can optimize across them.

Exception: If a CTE is referenced multiple times, OR if you use `MATERIALIZED`, it's materialized.

Example: Inline vs Materialized

```
WITH recent_orders AS (
  SELECT * FROM orders WHERE created_at > NOW() - INTERVAL '7 days'
)
SELECT * FROM recent_orders WHERE user_id = 123;
```

Postgres 12+: The optimizer inlines this and executes:

```
SELECT * FROM orders
WHERE created_at > NOW() - INTERVAL '7 days'
  AND user_id = 123;
```

It pushes the `user_id = 123` filter into the scan.

Forcing Materialization

If you want the old behavior (compute once, reuse):

```
WITH recent_orders AS MATERIALIZED (
  SELECT * FROM orders WHERE created_at > NOW() - INTERVAL '7 days'
)
SELECT * FROM recent_orders WHERE user_id = 123;
```

Now Postgres computes `recent_orders` once, stores it, then filters.

When to use MATERIALIZED:

- CTE is expensive and referenced multiple times
- You want to protect a carefully tuned subquery from being rewritten

When NOT to use MATERIALIZED:

- When you want the optimizer to push predicates down
- When the CTE is cheap or only used once

MySQL and Other Databases

- **MySQL 8.0+:** Has CTEs, but materialization behavior differs. Check docs.
- **SQL Server:** CTEs are inlined (like subqueries).
- **Oracle:** CTEs can be hinted with materialize/inline directives.

Window Functions: The Game Changer

Window functions let you compute aggregates **without collapsing rows**.

The Problem They Solve

Goal: Show each order and the user's total order count.

Without window functions (requires self-join):

```
SELECT
  o.id,
  o.total,
  user_counts.order_count
FROM orders o
JOIN (
  SELECT user_id, COUNT(*) AS order_count
  FROM orders
  GROUP BY user_id
) user_counts ON o.user_id = user_counts.user_id;
```

Messy.

With window functions:

```
SELECT
  id,
  total,
  COUNT(*) OVER (PARTITION BY user_id) AS order_count
FROM orders;
```

Magic: Each row keeps its identity, but you get aggregate info.

Window Function Syntax

```
<aggregate_function>() OVER (
  PARTITION BY <column>    -- Optional: groups (like GROUP BY, but non-collapsing)
  ORDER BY <column>        -- Optional: ordering within groups
  ROWS/RANGE <frame>       -- Optional: which rows to include in calculation
)
```

Key Window Functions

1. Aggregate Functions (COUNT, SUM, AVG, etc.)

```sql
SELECT
  name,
  salary,
  AVG(salary) OVER (PARTITION BY department_id) AS dept_avg_salary
FROM employees;
```

Result: Each employee sees their department's average salary, but rows aren't collapsed.

2. ROW_NUMBER(): Assign Row Numbers

```sql
SELECT
  name,
  salary,
  ROW_NUMBER() OVER (ORDER BY salary DESC) AS salary_rank
FROM employees;
```

Result:

```
name     | salary | salary_rank
---------|--------|------------
Alice    | 120000 | 1
Bob      | 110000 | 2
Charlie  | 100000 | 3
```

Use case: Ranking, pagination.

3. RANK() and DENSE_RANK(): Handle ties

RANK: Leaves gaps after ties.

```sql
SELECT
  name,
  score,
  RANK() OVER (ORDER BY score DESC) AS rank
FROM contestants;
```

Result:

```
name    | score | rank
--------|-------|-----
Alice   | 100   | 1
Bob     | 100   | 1    -- Tie
Charlie | 90    | 3    -- Gap!
```

DENSE_RANK: No gaps.

```
name    | score | dense_rank
--------|-------|------------
Alice   | 100   | 1
```

```
Bob     | 100   | 1
Charlie | 90    | 2   -- No gap
```

4. LAG() and LEAD(): Access Previous/Next Rows

LAG: Access previous row.

```
SELECT
  date,
  revenue,
  LAG(revenue) OVER (ORDER BY date) AS prev_day_revenue
FROM daily_sales;
```

Result:

```
date       | revenue | prev_day_revenue
-----------|---------|------------------
2024-01-01 | 1000    | NULL
2024-01-02 | 1200    | 1000
2024-01-03 | 1100    | 1200
```

LEAD: Access next row (same idea).

Use case: "Compare today's revenue to yesterday's."

5. FIRST_VALUE() and LAST_VALUE(): First/Last in Window

```
SELECT
  name,
  salary,
  FIRST_VALUE(name) OVER (PARTITION BY department_id ORDER BY salary DESC) AS
highest_paid
FROM employees;
```

Result: Each row shows who's the highest paid in their department.

PARTITION BY: Grouping Without Collapsing

```
SELECT
  product_id,
  sale_date,
  amount,
  SUM(amount) OVER (PARTITION BY product_id) AS total_product_sales
FROM sales;
```

What happens:

- Rows are logically grouped by `product_id`
- For each row, `total_product_sales` shows the sum for that product
- But all rows remain individual

Think of it as: "GROUP BY without the collapsing."

ORDER BY in Window Functions: Running Totals

```
SELECT
  sale_date,
  amount,
  SUM(amount) OVER (ORDER BY sale_date) AS running_total
FROM sales;
```

Result:

```
sale_date  | amount | running_total
-----------|--------|---------------
2024-01-01 | 100    | 100
2024-01-02 | 150    | 250
2024-01-03 | 200    | 450
```

Key insight: ORDER BY in OVER makes the aggregate cumulative.

Frame Clauses: ROWS and RANGE

By default, `ORDER BY` in a window function uses `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (running total).

You can customize:

ROWS BETWEEN

Sliding window of 3 rows:

```
SELECT
  date,
  amount,
  AVG(amount) OVER (ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS
moving_avg_3day
FROM sales;
```

Result: 3-day moving average (current + 2 previous).

Frame options:

- `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` : Last 3 rows
- `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` : From start to current (running total)
- `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` : Current row + 1 before + 1 after

RANGE BETWEEN

RANGE operates on values, not physical rows.

Example: "Average salary within $10k of current salary."

```sql
SELECT
  name,
  salary,
  AVG(salary) OVER (ORDER BY salary RANGE BETWEEN 10000 PRECEDING AND 10000 
FOLLOWING) AS nearby_avg
FROM employees;
```

ROWS vs RANGE:

- **ROWS:** Physical row positions
- **RANGE:** Logical value ranges

Real-World Window Function Examples

Example 1: Top 3 Products per Category

```sql
SELECT *
FROM (
  SELECT
    category_id,
    product_name,
    sales,
    ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY sales DESC) AS rank
  FROM products
) ranked
WHERE rank <= 3;
```

How it works:

1. Assign row numbers within each category, ordered by sales
2. Filter to top 3

Example 2: Month-over-Month Growth

```sql
SELECT
  month,
  revenue,
  LAG(revenue) OVER (ORDER BY month) AS prev_month,
  revenue - LAG(revenue) OVER (ORDER BY month) AS growth
FROM monthly_revenue;
```

Result:

```
month   | revenue | prev_month | growth
--------|---------|------------|-------
2024-01 | 10000   | NULL       | NULL
2024-02 | 12000   | 10000      | 2000
2024-03 | 11000   | 12000      | -1000
```

Example 3: Cumulative Sum (Running Total)

```sql
SELECT
  date,
  amount,
  SUM(amount) OVER (ORDER BY date) AS cumulative_revenue
FROM daily_sales;
```

Example 4: Latest Order Per User (Without Self-Join)

Remember this from the JOINs chapter?

Old way (self-join):

```sql
SELECT u.name, o.created_at
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.created_at = (
  SELECT MAX(created_at) FROM orders WHERE user_id = u.id
);
```

Window function way:

```sql
SELECT name, created_at
FROM (
  SELECT
    u.name,
    o.created_at,
    ROW_NUMBER() OVER (PARTITION BY u.id ORDER BY o.created_at DESC) AS rn
  FROM users u
  JOIN orders o ON u.id = o.user_id
) sub
WHERE rn = 1;
```

Cleaner, often faster.

When to Use Subqueries vs CTEs vs Window Functions

Use Subqueries When:

- Simple, one-off logic
- Inline filtering (EXISTS, IN)
- Scalar computations

Use CTEs When:

- Complex multi-stage queries
- Readability matters (almost always)
- Recursive logic
- Reusing the same logic multiple times

Use Window Functions When:

- You need aggregates but want to keep row detail

- Ranking, row numbering
- Running totals, moving averages
- Comparing rows to previous/next rows

Avoid:

- **Deep nesting of subqueries:** Use CTEs for readability
- **Correlated subqueries in tight loops:** Convert to JOINs or window functions
- **Overusing MATERIALIZED:** Let the optimizer do its job

Performance Considerations

Subquery Performance

Good:

```
-- Uncorrelated, runs once
WHERE user_id IN (SELECT id FROM active_users)
```

Bad:

```
-- Correlated, runs per row
WHERE total > (SELECT AVG(total) FROM orders WHERE user_id = outer.user_id)
```

Fix: Use JOIN or window function.

CTE Performance

Good:

```
WITH expensive_calc AS (
  SELECT ... -- Complex aggregation
)
SELECT * FROM expensive_calc
UNION ALL
SELECT * FROM expensive_calc;  -- Reuses result
```

Bad:

```
-- Forcing materialization when not needed
WITH trivial_filter AS MATERIALIZED (
  SELECT * FROM orders WHERE id = 123
)
SELECT * FROM trivial_filter;
```

Window Function Performance

Good:

```
-- Single pass
SELECT
```

```sql
  id,
  SUM(amount) OVER (ORDER BY date),
  AVG(amount) OVER (ORDER BY date)
FROM sales;
```

Bad (forces multiple sorts):

```sql
-- Different ORDER BY in each window = multiple sorts
SELECT
  id,
  SUM(amount) OVER (ORDER BY date),
  AVG(amount) OVER (ORDER BY product_id)
FROM sales;
```

Fix: Group compatible window functions together.

Pro Tip: EXPLAIN Your CTEs and Window Functions

Always check the execution plan. The optimizer might surprise you (good or bad).

```sql
EXPLAIN ANALYZE
WITH ...
```

Common Pitfalls

Pitfall 1: Forgetting PARTITION BY

Problem:

```sql
SELECT
  user_id,
  order_id,
  ROW_NUMBER() OVER (ORDER BY created_at) AS order_num
FROM orders;
```

Bug: Row numbers are global, not per user.

Fix:

```sql
ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY created_at)
```

Pitfall 2: Using LAST_VALUE Without Frame Clause

Problem:

```sql
SELECT
  date,
  amount,
```

```
  LAST_VALUE(amount) OVER (ORDER BY date) AS last_amount
FROM sales;
```

Bug: `last_amount` is just `amount` (current row is the last in the default frame).

Fix:

```
LAST_VALUE(amount) OVER (ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING)
```

Or just use `FIRST_VALUE` with `DESC`.

Pitfall 3: Filtering on Window Functions in WHERE

Problem:

```
SELECT
  name,
  ROW_NUMBER() OVER (ORDER BY salary DESC) AS rank
FROM employees
WHERE rank <= 5;  -- ERROR!
```

Why it fails: Window functions are evaluated after WHERE.

Fix: Use a subquery:

```
SELECT * FROM (
  SELECT
    name,
    ROW_NUMBER() OVER (ORDER BY salary DESC) AS rank
  FROM employees
) sub
WHERE rank <= 5;
```

Key Takeaways

1. **Subqueries let you nest logic.** Prefer uncorrelated over correlated.
2. **CTEs improve readability.** Use them for complex multi-stage queries.
3. **Recursive CTEs solve hierarchical problems.** Watch for infinite loops.
4. **Window functions keep row detail while computing aggregates.** Massive power.
5. **PARTITION BY is like GROUP BY without collapsing rows.**
6. **ORDER BY in window functions enables running totals and LAG/LEAD.**
7. **CTEs in Postgres 12+ are inlined unless materialized or referenced multiple times.**
8. **Always EXPLAIN complex queries** to see what the optimizer is doing.

Next up: GROUP BY and aggregations in depth—because that's where things get weird.