Subqueries, CTEs, and Window Functions

Scalar and correlated subqueries, chained WITH (CTEs), and window functions (ROW_NUMBER, RANK, PARTITION BY) -- computing across rows without collapsing them the way GROUP BY does, with this project's own real topic table. The 1st lesson in the Advanced PostgreSQL category.

Advanced 30 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/subqueries-ctes-and-window-functions

---

PostgreSQL Foundations built up the everyday SQL toolbox — `SELECT`, `JOIN`, `GROUP BY`. Advanced PostgreSQL, starting here, is where PostgreSQL-specific depth concentrates. Subqueries, CTEs, and window functions share one idea: computing something across a set of rows *without* collapsing them into fewer rows the way `GROUP BY` does — each keeps every original row intact while still answering a question that needs to look beyond just that one row.

Subqueries: A Query Inside a Query, Revisited

"Inserting, Updating, and Deleting Data" and "JOINs" already used subqueries — a `SELECT` nested inside another statement — to look up an id by slug. A subquery can also stand in for a single value used in a comparison:

```
SELECT slug, estimated_minutes
FROM topic
WHERE estimated_minutes > (SELECT AVG(estimated_minutes) FROM topic);
```

The parenthesized `SELECT` here is a **scalar subquery** — it must return exactly one row and one column, since it's being compared to `estimated_minutes` with `>` the same way a literal number would be. Run against this project's own real data, this returns every topic that takes longer than the platform-wide average — genuinely useful, and something no single `WHERE` condition without a subquery could express, since the average itself depends on scanning the whole table first.

Correlated Subqueries

The subquery above computes one number, entirely independent of the outer query's rows. A **correlated subquery** is different — it references a column from the outer query, and is re-evaluated once per outer row:

```
SELECT t.slug, t.estimated_minutes
FROM topic t
WHERE t.estimated_minutes > (
    SELECT AVG(t2.estimated_minutes)
    FROM topic t2
    WHERE t2.category_id = t.category_id
);
```

This is a real, meaningfully different question from the one above: not "longer than the platform-wide average," but "longer than the average *for its own category*" — `t2.category_id = t.category_id` is what makes it correlated, recomputing the inner average separately for every category a row in the outer query belongs to. Correlated subqueries are powerful but can be slow on large tables, since the inner query conceptually reruns per outer row — "Indexes and Query Performance with EXPLAIN," later in this category, comes back to reasoning about exactly this kind of cost.

WITH: Common Table Expressions (CTEs)

A `WITH` clause names a subquery up front, letting the main query reference it like a real table — useful the moment a query needs to reason about the same intermediate result more than once, or simply reads better broken into named steps:

```
WITH category_counts AS (
    SELECT category_id, COUNT(*) AS topic_count
    FROM topic
    GROUP BY category_id
)
SELECT cat.name, cc.topic_count
FROM category cat
JOIN category_counts cc ON cc.category_id = cat.id
ORDER BY cc.topic_count DESC;
```

`category_counts` isn't a real table anywhere in this project's schema — it exists only for the duration of this one query, computed once by the `WITH` clause and then queried like any other table in the `SELECT` that follows. This is genuinely the same aggregation "Aggregation and GROUP BY" already built (topics per category), now given a name and joined against `category` in the same statement instead of computed as a one-off.

A Real Multi-Step CTE Example

CTEs can chain, each one building on the last — useful for expressing a multi-step calculation as readable stages rather than one deeply nested query:

```
WITH category_counts AS (
    SELECT category_id, COUNT(*) AS topic_count
    FROM topic
    GROUP BY category_id
),
above_average AS (
    SELECT category_id, topic_count
    FROM category_counts
    WHERE topic_count > (SELECT AVG(topic_count) FROM category_counts)
)
SELECT cat.name, aa.topic_count
FROM category cat
JOIN above_average aa ON aa.category_id = cat.id;
```

`above_average` references `category_counts`, the CTE defined just before it — each stage reads like an English sentence ("count topics per category," then "keep only the above-average ones"), where the equivalent single query, with the aggregation nested inside a subquery inside another subquery, would be considerably harder to read in one pass.

Window Functions: Computing Across Rows Without Collapsing Them

`GROUP BY`, already covered, always reduces many rows to fewer — one row per group. A **window function** computes an aggregate-like value *per row*, while keeping every original row in the result:

```
SELECT slug, category_id, estimated_minutes,
       AVG(estimated_minutes) OVER (PARTITION BY category_id) AS category_avg
FROM topic;
```

Every single `topic` row still appears in the output — nothing is collapsed — but each one now also carries its own category's average alongside it. `OVER (...)` is what marks `AVG(...)` here as a window function rather than a regular aggregate; without it, this would need a `GROUP BY` and would lose every column not listed in it.

ROW_NUMBER(), RANK(), and PARTITION BY

Beyond aggregates used as window functions, PostgreSQL has functions that only make sense as window functions — `ROW_NUMBER()` and `RANK()` among the most common:

```
SELECT slug, category_id, estimated_minutes,
       ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY estimated_minutes DESC) AS
rank_in_category
FROM topic;
```

`PARTITION BY category_id` divides the rows into independent groups, exactly like `GROUP BY` would for aggregation — but instead of collapsing each group to one row, `ROW_NUMBER()`

numbers the rows *within* each partition, restarting at `1` for every new `category_id`, in the order `ORDER BY estimated_minutes DESC` establishes. `RANK()` behaves almost identically, except tied values (two topics with the identical `estimated_minutes`) receive the same rank, with the next rank after them skipping ahead accordingly (a rank of `1, 1, 3`, not `1, 1, 2`) — `ROW_NUMBER()` never ties, always producing a strict `1, 2, 3`.

A Real Window Function Example

Run against this project's own `postgresql-foundations` category, the `ROW_NUMBER()` query above surfaces something genuinely worth knowing: which topic in that category takes the longest, which the shortest, and everything in between — a ranking `GROUP BY` alone has no way to produce, since it can only report one summary row per category, never a per-row position within it. This is the concrete difference window functions add: an aggregate answers "what's true about the group as a whole"; a window function answers "where does *this* row stand within its group," while never losing sight of the row itself.

Common Misconceptions

"A CTE is always faster than the equivalent subquery." Not necessarily — a `WITH` clause is primarily a readability and reusability tool; PostgreSQL is free to optimize a CTE and an equivalent nested subquery similarly in many cases, and the performance characteristics genuinely depend on the query. **"Window functions and `GROUP BY` do the same thing, just with different syntax."** They don't — `GROUP BY` reduces row count; a window function never does, which is precisely why the two solve different problems even when both start from an aggregate like `AVG`. **"`RANK()` and `ROW_NUMBER()` are interchangeable."** Only when there are no ties in the `ORDER BY` column — the moment two rows share a value, they diverge, and picking the wrong one silently produces a subtly wrong ranking.

Best Practices

- Reach for a correlated subquery only when the condition genuinely needs to reference the outer row (like "above this category's own average") — an uncorrelated scalar subquery, computed once, is simpler and normally cheaper whenever the value doesn't depend on the outer row at all.
- Break a multi-step calculation into named CTEs, as this lesson's "above-average categories" example does, rather than nesting subqueries several levels deep — it costs nothing at the SQL level and reads far more like the English description of the problem.
- Choose `ROW_NUMBER()` when a strict, tie-free ordering is required (like "assign exactly one rank to each row"), and `RANK()` specifically when ties should share a position — picking between them is a decision about ties, not a stylistic one.

- Add `PARTITION BY` to a window function the moment "per row" should really mean "per row within its group" (per category, per difficulty) — omitting it treats the entire result as one partition, silently changing what the computed value means.

Common Mistakes

- Writing a scalar subquery that can actually return more than one row, and hitting a genuine runtime error ("more than one row returned by a subquery used as an expression") only when the data happens to produce a second matching row.
- Referencing a later CTE from an earlier one — CTEs (without `RECURSIVE`) can only reference the ones defined before them in the same `WITH` clause, in order, the same top-to-bottom dependency direction as reading the query itself.
- Forgetting `PARTITION BY` on a window function that was meant to compute "per category" or "per difficulty," and getting a single value computed across the entire table instead — the query runs without error, silently answering a broader question than intended.
- Assuming a window function reduces row count the way an aggregate with `GROUP BY` would — it never does; a query expecting fewer output rows after adding `OVER (...)` will be surprised to see every original row still present.

Summary, Cheat Sheet, and Glossary

Summary

- A scalar subquery returns exactly one value and can be used anywhere a literal would be; a correlated subquery references the outer query's row and is re-evaluated per outer row — this project's own "topics longer than their category's average" is a real correlated example.
- `WITH <name> AS (...)` defines a CTE, a named, one-use-per-query subquery that can be referenced like a table in the statement that follows — CTEs can chain, each one building on the last.
- Window functions (`OVER (...)`) compute an aggregate-like or ranking value per row without collapsing the result, unlike `GROUP BY` — every original row survives.
- `PARTITION BY` divides rows into independent groups for a window function, the window-function equivalent of `GROUP BY`'s grouping; `ORDER BY` inside `OVER (...)` establishes the order `ROW_NUMBER()` / `RANK()` count through.
- `ROW_NUMBER()` always produces a strict, tie-free sequence; `RANK()` gives tied rows the same rank and skips ahead afterward — the choice between them is about how ties should be handled, not a stylistic preference.

Cheat Sheet

```
-- Scalar subquery
SELECT a FROM t WHERE a > (SELECT AVG(a) FROM t);

-- Correlated subquery
SELECT a FROM t t1 WHERE a > (SELECT AVG(a) FROM t t2 WHERE t2.g = t1.g);

-- CTE
WITH x AS (SELECT g, COUNT(*) AS c FROM t GROUP BY g)
SELECT * FROM x WHERE c > 1;

-- Window functions
SELECT a, AVG(a) OVER (PARTITION BY g) FROM t;
SELECT a, ROW_NUMBER() OVER (PARTITION BY g ORDER BY a DESC) FROM t;
SELECT a, RANK() OVER (PARTITION BY g ORDER BY a DESC) FROM t;
```

Glossary

- **Scalar subquery**: a subquery that must return exactly one row and one column, usable anywhere a single value is expected.
- **Correlated subquery**: a subquery that references a column from the outer query, re-evaluated once per outer row.
- **CTE (Common Table Expression)**: a named subquery, introduced with `WITH`, referenceable like a table for the rest of the statement.
- **Window function**: a function computed per row over a defined set of related rows (its "window"), without reducing the number of output rows the way `GROUP BY` does.
- **PARTITION BY**: divides rows into independent groups for a window function, analogous to `GROUP BY` for aggregation.