Engineer

CTE

a named subquery a bigger query reads as a table

What it is

A CTE, short for common table expression, is a query given a name in a WITH clause at the top of a statement, so that the rest of the statement can read its rows as though they came from a table. A statement can define several CTEs, each one able to read the ones named before it, and the final query at the bottom puts them together. PostgreSQL's manual describes them as "temporary tables that exist just for one query", which is the right picture with one caveat: nothing is stored under that name, and the next statement cannot see it. Each CTE can be a SELECT, or in PostgreSQL an INSERT, UPDATE, DELETE or MERGE, and the statement the WITH clause is attached to can be any of those as well.

A question such as "which meters used more than three times their own usual amount on some day in the last 30 days" splits naturally into steps:

```
WITH daily AS (
  SELECT meter_id, read_at::date AS day, sum(kwh) AS kwh
  FROM readings
  WHERE read_at >= current_date - 30
  GROUP BY meter_id, read_at::date
),
typical AS (
  SELECT meter_id, avg(kwh) AS avg_kwh
  FROM daily
  GROUP BY meter_id
)
SELECT d.meter_id, d.day, d.kwh, t.avg_kwh
FROM daily AS d
JOIN typical AS t USING (meter_id)
WHERE d.kwh > 3 * t.avg_kwh
ORDER BY d.day, d.meter_id;
```

The first step totals each meter's readings per day, the second averages those totals per meter, and the last compares every day against its own meter's average. The same answer can be written with nested subqueries, but the daily totals would then appear twice in the text, once inside the other, and a reader would have to work from the inside out. With WITH, the logic reads from top to bottom, and each step can be run on its own while it is being checked.

Adding RECURSIVE lets a CTE read its own output, which is the part plain SQL has no other way to express. A recursive CTE has a starting query, then UNION or UNION ALL, then a query that refers back to the CTE. PostgreSQL runs the starting query, keeps its rows in a working table, and keeps running the second query against the newest rows until a round adds none. That is how a statement walks a tree of unknown depth, such as the categories under one heading:

```
WITH RECURSIVE subtree AS (
  SELECT id, parent_id, name, 1 AS depth
  FROM categories
  WHERE id = 42
  UNION ALL
  SELECT c.id, c.parent_id, c.name, s.depth + 1
  FROM categories AS c
  JOIN subtree AS s ON c.parent_id = s.id
)
SELECT id, name, depth FROM subtree ORDER BY depth, name;
```

Use the concept in a decision

Put "What is a CTE?" to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

The first reason is legibility. An analytical query of a hundred lines that nests four levels deep is hard to review and harder to change. The same logic as a chain of named steps gives a reviewer a sentence per step, gives an author a place to put a comment, and lets anyone run one step alone to see whether its rows look right before trusting the next.

The second reason is how the database evaluates the steps, and here the details changed over time. PostgreSQL normally evaluates a WITH query once per execution of the statement, even when the statement refers to it more than once, so an expensive calculation needed in two places runs once. The cost of that guarantee is that the planner cannot push a filter from the outer query into a CTE that is referenced several times, because the filter should apply to one use and not the others. The CTE is computed in full and the rows are thrown away afterwards.

Until PostgreSQL 12, that was true of every CTE: each was evaluated before the rest of the query and never merged into it. Version 12 changed the default. A CTE that is not recursive, has no side effects and is referenced only once is now folded into the outer query, so the planner optimises the two levels together and can carry the outer filter down to an index on the base table. MATERIALIZED forces the old behaviour, and NOT MATERIALIZED forces folding even for a CTE used more than once, at the risk of running its work twice. SQLite accepts the same two keywords as non-binding hints and says it copied the idea from PostgreSQL. Other engines decide for themselves, and the plan is where to check.

The third reason is that a CTE can change data. A DELETE with RETURNING in a WITH clause can pass its output to an INSERT in the main statement, which moves rows from one table to another in a single statement that either happens in full or not at all. Data-modifying CTEs run exactly once and always to completion, whether or not anything reads their output, which makes them a tool for doing several writes together rather than a way of describing a result.

What it looks like in practice

The work behind Accelerated PostgreSQL performance by 10x was a database for user accounts, tenants, locations and daily electricity readings that slowed as the data grew. The fix came in three parts: targeted and GiST indexes, time-based partitioning of the readings table, and query rewrites. The rewrites used CTEs and materialized views for frequently accessed data to avoid full table scans, with EXPLAIN ANALYZE pointing at the joins and subqueries worth restructuring. The tenfold improvement in response times and the 40% drop in resource use came from the three parts together, not from CTEs alone. The pairing is worth noting, because the two do different jobs: a CTE names an intermediate result for the length of one statement, while a materialized view stores a result that many statements can read until it is refreshed.

The other side of CTEs, legibility, carried the 500 electricity and GIS data analysis reports. Every report was written in pure SQL against PostgreSQL and BigQuery over smart-meter data with coordinates, meter IDs and timestamped usage, and the scripts used CTEs together with window functions, subqueries and geospatial joins. Each report shipped with annotated SQL so that colleagues could reproduce and audit the analysis, plus notes on common data-quality issues such as missing GPS coordinates and corrupted meter values. SQL that reads step by step travels well: the reports became a standard reference across departments and helped make operational planning 10–15% more efficient. Named steps are what make SQL of that kind annotatable at all: a note can sit above each step and say what it removes or assumes.

Where it goes wrong

A filter applied after all the work is done. A CTE referenced twice, or marked MATERIALIZED, is computed in full before the outer query filters it, and the copy it produces has no indexes. A query that should touch a few rows reads a whole table. Read the plan: when the CTE appears as its own step, with row counts far above what the query returns, try NOT MATERIALIZED or restate the query so the CTE is referenced once. Keep the materialised form where the CTE calls an expensive function, so that the function runs once per row and not once per reference.

A recursive query that keeps going. Data with a cycle, such as two categories each listed as the other's parent, sends a recursive CTE with UNION ALL round the loop for as long as the server lets it. UNION helps only when whole rows repeat, which rarely holds once a depth column is added. Track the path of visited keys in an array and stop when a key reappears, or use the CYCLE clause that PostgreSQL 14 added. A LIMIT on the outer query works as a testing trick, and the manual advises against relying on it in production. SQL Server caps recursion at 100 levels by default and cancels the statement beyond that, so the same bug stops instead of running on.

Changes the rest of the statement does not see. A data-modifying CTE and the main query run concurrently on the same snapshot, so a main query that reads the target table directly sees the rows as they were before the change. The only channel between them is RETURNING. Two parts of one statement that modify the same row are worse: only one modification takes effect, and which one is hard to predict. Read changed rows from the CTE's RETURNING output, and keep each row to a single writer per statement.

A plan that moved with an upgrade. Code written for PostgreSQL 11 or earlier sometimes relied on a CTE being evaluated first, as a deliberate barrier against a plan the author did not want. After an upgrade to 12 or later, the same CTE is folded into the outer query and the planner picks something new, usually faster and occasionally much slower. Compare EXPLAIN output for the slow statements before and after the upgrade, and add MATERIALIZED where the barrier was intended.

Further reading

- 7.8. WITH Queries (Common Table Expressions) — the manual's chapter on CTEs: recursion, cycle detection, materialisation and data-modifying statements.
- E.23. Release 12 — the release that started folding single-use CTEs into the outer query.
- E.25. Release 14 — the release that added the SQL-standard SEARCH and CYCLE clauses.
- The WITH Clause — SQLite's version, including its materialisation hints and where it borrowed them from.
- WITH common_table_expression (Transact-SQL) — SQL Server's rules for CTEs, including the MAXRECURSION limit on recursion.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a CTE?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Printed from https://engineer.company/glossary/cte/

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.