Engineer

Query

a written question a database answers

What it is

A query is a request for data, written in a language the database understands, that comes back with a result: usually a set of rows, sometimes a single number. PostgreSQL's manual defines it in one line: retrieving data from a database, or the command that does the retrieving, is called a query, and in SQL that command is SELECT. Everyday usage stretches the word further, and engineers call almost any statement they send to a database a query, including the ones that change data and hand something back. Both senses appear below, and the plural, queries, usually means all the traffic an application sends.

Asking which invoices have waited longest for payment reads close to how the question would be put aloud:

```
SELECT i.number, c.name, i.amount, i.due_on
FROM invoices AS i
JOIN customers AS c ON c.id = i.customer_id
WHERE i.paid_on IS NULL
  AND i.due_on < current_date
ORDER BY i.due_on
LIMIT 20;
```

The text names what it wants and leaves the method open. Between receiving it and returning the first row, PostgreSQL's planner lists the ways it could get the answer, such as reading the whole invoices table or walking an index on `due_on`, estimates a cost for each, and runs the cheapest. Those costs are in arbitrary units, conventionally anchored to reading one page from disk. Two queries that look alike on screen can therefore differ by a factor of a thousand once the data under them grows, and the plan is the only place that difference shows.

The clauses do not run in the order they are written. The manual describes the processing of SELECT as WITH first, then FROM, WHERE, grouping and HAVING, then the output list, and only after that ORDER BY and LIMIT. That order explains a rule that surprises most people once: a name given to a column in the select list works in ORDER BY and GROUP BY, while WHERE runs before the name exists, so there the expression has to be written out again. The result of every query is itself a table, which is why queries nest, with a subquery or a WITH clause feeding its rows into a larger one.

Use the concept in a decision

Put “What is a query?” 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

In an application backed by a relational database, the queries decide much of how fast a page feels and how much hardware the system needs. Which queries matter is arithmetic rather than instinct: a query’s cost to the system is its duration multiplied by how often it runs. A 2 ms lookup run a million times an hour adds up to more than half an hour of database time every hour, while a report that takes two seconds and runs twice a day barely registers.

PostgreSQL keeps that tally in the pg_stat_statements extension. It folds statements that differ only in their literal values into one entry and records how many times each ran and the total time it took. Sorting by the total rather than the average is the habit that pays off first in query work, because it points at the statement worth fixing instead of the one that only looks slow.

Every query also costs a round trip. The application sends text, the server parses, plans and runs it, and rows travel back. For a quick lookup on an index, the trip can outweigh the work. So the shape of the conversation matters as much as any single statement: one UPDATE that touches a hundred rows makes one trip, and a loop that reads a hundred rows and updates each in turn makes a hundred and one.

Then there is the question of who writes the queries. An ORM or a query builder generates them from application code, which is quick for simple reads and writes and keeps the code in one language. It also hides how many statements a request sends and what they look like, until a page turns slow. Hand-written SQL shows both, at the price of someone owning it like any other code. A workable split lets the ORM handle routine records, writes the hot paths by hand, and reads the generated SQL of anything that climbs to the top of the totals.

What it looks like in practice

The method for a slow database in Optimise the database before you shard it starts with queries, before anyone prices more machines. The first step asks PostgreSQL which statements cost the most in total, using pg_stat_statements sorted by total execution time. The second runs EXPLAIN (ANALYZE, BUFFERS) on the statement at the top, to see the plan the database chose, the rows each step really returned against the rows the planner expected, and how many pages it read. The note's rule of thumb is that a scan of a million rows to return ten, or an estimate a hundred times off, usually explains the slowness on its own. It counts round trips too, and draws the line this article draws above: whatever has to see the data moves into the database, and business rules stay in the application.

A query can also do the work of a lock. In Selling the last ticket exactly once a flash sale goes wrong in one of two ways. Reading the stock level, deciding in the application and writing the new number back lets two buyers both take the last unit; holding a row lock across that application logic makes every buyer wait behind it. The fix folds the check and the decrement into one conditional UPDATE that subtracts the quantity only where enough stock remains and returns the new level. One row back means the unit is reserved, and zero rows means sold out. At READ COMMITTED, PostgreSQL's default isolation level, a second buyer who reaches the row while the first holds it waits, then evaluates the WHERE clause again against the committed row, so the last unit goes to exactly one of them. A CHECK constraint that keeps stock at zero or above stands behind the query.

Where it goes wrong

A cheap query that dominates the load. The database runs hot, yet the slow-query log stays quiet, because each call finishes in a few milliseconds. The cause is frequency: one fast query called on every request, often from a loop or from a

component that appears on every page. Rank statements by total execution time in pg_stat_statements, then cache the result, batch the calls, or make the query cheaper with an index that matches its filter.

A page that sends hundreds of queries. Each query in the trace takes a millisecond, and the page still takes seconds. An ORM loading related records lazily, one query per parent row, is the usual cause, known as the N+1 pattern. Load the related rows in one query, with a join or with `WHERE id = ANY($1)` and an array of keys, and watch the count of statements per request as closely as the duration of any one.

The last unit sold twice. Two requests read the same stock level, both decide there is enough, and both write. Reading and writing in separate statements leaves a gap between the check and the action, and concurrent traffic finds it. Close the gap inside the query, with a conditional UPDATE and RETURNING as above, or with SELECT … FOR UPDATE held as briefly as possible and with no network call between the lock and the commit.

A diagnosis that changed the data. Someone runs EXPLAIN ANALYZE on a slow DELETE to look at its plan, and the rows are gone. ANALYZE executes the statement for real and discards only its output, so every side effect happens. Wrap any data-modifying statement in BEGIN and ROLLBACK before analysing it, or use plain EXPLAIN, which plans without executing, when estimates are enough.

Further reading

- SELECT — the full syntax of a query in PostgreSQL, including the order in which its clauses are processed.
- 7.1. Overview — the start of the manual's chapter on queries, with the definition of the word.
- 51.1. The Path of a Query — the trip from the client's text through parser, rewriter and planner to the executor.
- 14.1. Using EXPLAIN — how to read a plan, what the cost units mean, and why ANALYZE runs the statement.
- F.32. pg_stat_statements — track statistics of SQL planning and execution — the extension that ranks queries by what they cost in total.
- 13.2. Transaction Isolation — what a second writer sees under READ COMMITTED, and why the WHERE clause is checked again.

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 query?” 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

Carefully Crafted With Love by D.

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

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

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