Engineer

PostgreSQL

the database most of this work was built on

What it is

PostgreSQL is a relational database server, free and open source, that keeps data in tables, answers questions written in SQL and makes every transaction all-or-nothing. Most people who work with it call it Postgres, and the project treats both names as its own. Asked “what is PostgreSQL?”, a working engineer would put it plainly: it is the general-purpose database a new system starts on when nobody has a specific reason to choose another.

It is older than most of the software it now sits underneath. The POSTGRES project began at the University of California at Berkeley under Professor Michael Stonebraker, and its implementation started in 1986. Andrew Yu and Jolly Chen gave it an SQL interpreter in 1994 and released the result as Postgres95. Two years later it was renamed PostgreSQL, and the version numbers restarted at 6.0 to continue the sequence Berkeley had begun. The project itself describes the system as object-relational: beyond tables and joins, a user can add new data types, functions, operators, aggregates, index methods and procedural languages, and the server treats them as its own.

Architecturally it is a client and a server. One server process owns the data files, and every client connection gets a process of its own, forked for that connection alone. Applications send it SQL over that connection, through a driver for whatever language they are written in. Concurrency rests on multiversion concurrency control: an update writes a new version of a row rather than overwriting the old one, so each statement reads a consistent snapshot, readers never wait for writers, and writers never wait for readers.

Where does it sit among the alternatives? Beside MySQL and MariaDB as the other open-source relational servers, beside Oracle and Microsoft SQL Server as the commercial ones, beside SQLite as the embedded database that lives inside

one program, and against document stores such as MongoDB, which keep records without a fixed schema.

Use the concept in a decision

Put “What is PostgreSQL?” 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

Choosing a database is a decision a team lives with for years, because the data outlasts the code written against it. PostgreSQL is the low-regret default for three reasons a buyer can check.

The first is cost and lock-in. It is released under the PostgreSQL License, a short permissive licence in the family of BSD and MIT, so there is no per-core fee, no audit and no vendor who can change the terms. Amazon, Microsoft and Google each sell it as a managed service, and the same database runs on a laptop, which means moving between hosts is a migration of data rather than a rewrite of the application.

The second is breadth. One server covers relational tables, JSON documents in `jsonb` columns, full-text search, geospatial queries through the PostGIS extension, and scheduled jobs through `pg_cron`. Each of those is a separate product a team does not have to run, secure, back up and hire for. For most products, PostgreSQL versus MySQL is decided here: for plain reads and writes either serves well, and the difference shows when the work needs richer types, stricter constraints or an extension.

The third is longevity. The project ships one major version a year and supports each for five years, so an upgrade can be planned for instead of forced by an emergency. PostgreSQL 18, released in September 2025, added asynchronous I/O and a native function for time-ordered UUIDs, and SQL written for releases a decade older generally runs on it unchanged.

The trade-offs are real. A process per connection makes connections expensive, so a busy application needs a pooler. It grows on one larger machine more easily than across several: spreading writes over many servers means partitioning, replication or an extension, and each adds operational weight. And its defaults are tuned for a small machine, so a serious installation has to be configured on purpose.

What it looks like in practice

Many of the projects on this site run on PostgreSQL, and three of them show three different ways to lean on it.

On one platform the database became the whole data layer. Every read and write goes through one of 1,275 PostgreSQL functions across 34 schemas, and the Go API has no direct access to any table. The role the API logs in as, `mariner`, holds EXECUTE on the functions and USAGE on the schemas and nothing else, roughly 4,000 explicit grants. The functions run as SECURITY DEFINER under a separate non-login owner with a pinned `search_path`. The access boundary is enforced by Postgres itself; the price is that a new field means a migration and a function change.

On an application holding tenants, locations and daily electricity readings, the work was tuning rather than design, and the brief was 10x without overhauling the architecture. GiST indexes for the spatial data, composite indexes on tenant and date range, range partitioning of the electricity table by day and month, and queries rewritten after reading their plans with `EXPLAIN ANALYZE` improved query response times by 10x and cut resource utilization by 40%.

And on a platform with slow background work, the job queue lives in the same database. River keeps its queue in PostgreSQL, so a job and the rows it touches commit or roll back together. Behind it are 15 worker modules, 8 scheduled tasks and 20 `pg_cron` jobs that prune partitions, rotate salts and refresh aggregates. The page states the limit plainly: a queue in Postgres stops being the right answer at some volume, and one less moving part was worth more than headroom nobody was going to use.

Where it goes wrong

PostgreSQL rarely loses data. It does surprise people, and the surprises come in a few familiar shapes.

The table grows while the row count stays flat. An `UPDATE` or `DELETE` leaves the old row version behind until `VACUUM` removes it, because another transaction may still be reading it. A transaction left open for hours holds that cleanup back for the whole database, and autovacuum's defaults suit small tables. Close idle transactions, tune autovacuum per table for the large ones, and partition data that expires so that old months are dropped rather than deleted. Vacuum also does a second job: transaction IDs are 32 bits, every table has to be vacuumed at least once every two billion transactions, and a server that falls too far behind refuses new writes to protect the data.

Memory runs out at the first traffic spike. Each connection is a server process with its own memory, and the default limit is 100 connections. Twenty application instances with a pool of fifty each ask for a thousand processes. Put a pooler such as PgBouncer in front, and size each application's pool for the work, which is usually a handful of connections per instance.

A query that took milliseconds starts taking seconds. The planner chose a different plan as the data grew, often because its statistics went stale or the indexes were built for the queries of a year ago. Read the plan before changing anything, refresh statistics with `ANALYZE`, and index the predicate the query actually filters on:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, total FROM orders
WHERE customer_id = 42 AND placed_at >= '2026-01-01';
```

Inserts slow down as the table gets large. Random UUIDs as primary keys land all over the B-tree index, so every insert touches a different page and the index fragments. Time-ordered identifiers avoid this: one platform made UUID v7 its default key through PostgreSQL 18's `uuidv7()`, and new rows sort into the index instead of scattering across it.

Further reading

- What Is PostgreSQL? — the project's own definition, with the features and extension points it names.
- A Brief History of PostgreSQL — Berkeley, Postgres95 and the rename, from the people who did it.

- Concurrency Control: Introduction — how snapshots let reads and writes proceed without blocking each other.
- Routine Vacuuming — dead row versions, autovacuum and transaction ID wraparound in full.
- Using EXPLAIN — how to read a query plan, with worked examples.
- Versioning Policy — the yearly major release and the five years of support behind each.

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 PostgreSQL?” 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/postgresql/