PostgreSQL Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What PostgreSQL is 3
Installing it 5
psql 11
Connecting, and ”could not connect” 16
Clusters, databases and schemas 18
Roles and permissions 22
Types worth knowing 26
Identity columns, sequences and constraints 33
RETURNING and upserts 38
JSONB 41
Full-text search 47
EXPLAIN ANALYZE 53
Indexes 59
Transactions, MVCC and locks 65
Node.js with pg 74
Migrations without downtime 81
Backups and restores 84
Upgrading to a new major version 88
Health checks and where to host it 91

Preface

This is the PostgreSQL Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

This book is about PostgreSQL itself: the server, how you connect to it, who can do what, the types and features you won't find in other databases, how it handles many writers at once, and how you keep it fast and backed up.

It assumes you know SQL. If `SELECT`, `JOIN` and `GROUP BY` are new to you, start with my SQL Handbook, which teaches the language on SQLite, PostgreSQL and MySQL side by side. This book picks up where that one stops.

We use the same real database as the SQL Handbook: every post I published on my blog, 2,193 of them, with their tags. This time I also load the text of each post (about 2 million words in total) and the metadata at the top of each file as JSON. That's enough data for a full-text search engine, and for indexes that make a measurable difference.

Every command and every output in this book comes from a real run, in November 2026, on these versions:

Software	Version
PostgreSQL	18.7 (Homebrew on macOS)
PostgreSQL on Ubuntu	18.7 (PGDG packages, Ubuntu 24.04)
PostgreSQL 17, for the upgrade chapter	17.11
Node.js	22.23.4
pg (Node.js driver)	8.23.1

Where something changed in PostgreSQL 18, I say so. If you run an older version, most of the book still applies, and I point out the parts that don't.

What PostgreSQL is

PostgreSQL, or Postgres, is a relational database server. It's open source under a permissive license close to MIT and BSD, it's been in development since 1986, and today it's the default choice for a new application that needs a database on a server.

The key word is **server**.

SQLite, which I used for most of the SQL Handbook, is a library: your program opens a file and reads it. PostgreSQL is a separate program that owns the data. Your application connects to it, over a network socket or a local one, logs in as a user, and sends SQL. The server runs the query and sends back the result.

That's more setup than a file. In exchange you get many clients writing at the same time, users with different permissions, a server you can put on another machine, and backups you can take while everything keeps running.

The processes

When PostgreSQL runs, it's not one process. Here's what I see on my Mac with the test server I use for this book:

```
88002 /opt/homebrew/Cellar/postgresql@18/18.7/bin/postgres -D pgdata -p 5418 -k /tmp/pgb
88005 postgres: io worker 0
88006 postgres: io worker 1
88007 postgres: io worker 2
88008 postgres: checkpointer
88009 postgres: background writer
88011 postgres: walwriter
88012 postgres: autovacuum launcher
88013 postgres: logical replication launcher
```

The first one is the main server process, the **postmaster**. It listens for connections and starts everything else.

The others do the background work:

- the **checkpointer** and the **background writer** move changed data from memory to the data files
- the **walwriter** writes the write-ahead log, which we'll meet in the backups chapter
- the **autovacuum launcher** starts workers that clean up old row versions (the MVCC chapter explains why that's needed)
- the **io workers** are new in PostgreSQL 18, which reads data from disk asynchronously

Every time a client connects, the postmaster starts one more process just for that connection. A hundred connected clients means a hundred processes. Keep that in mind, because it's why connection pools exist, and we'll build one in the Node.js chapter.

The data directory and the cluster

All the data lives in one folder, the **data directory**. This is mine:

```
PG_VERSION    pg_dynshmem   pg_multixact  pg_snapshots  pg_tblspc     pg_xact
base          pg_hba.conf   pg_notify     pg_stat       pg_twophase   postgresql.auto.conf
global        pg_ident.conf pg_replslot   pg_stat_tmp   pg_wal        postgresql.conf
pg_commit_ts  pg_logical    pg_serial     pg_subtrans   postmaster.opts
```

You never edit the files in it by hand, except the configuration files: `postgresql.conf` for the server settings and `pg_hba.conf` for who can connect.

One server process with one data directory is called a **cluster**. It's a confusing name, because it has nothing to do with a cluster of machines. A cluster is one PostgreSQL installation that holds several **databases**.

A fresh cluster already has three:

- `postgres`, an empty database to connect to when you don't have one yet
- `template1`, which every new database is copied from

- `template0`, a pristine copy of `template1` in case you break it

Each database is a folder inside `base/`, named after its internal ID:

```
SELECT oid, datname FROM pg_database ORDER BY oid;
```

```
  oid  |  datname
-------+-----------
     1 | template1
     4 | template0
     5 | postgres
 16384 | blog
(4 rows)
```

And `ls pgdata/base` shows exactly those four folders: `1`, `4`, `5` and `16384`.

Why PostgreSQL

I use PostgreSQL when an application has more than one writer, or will live on a server, or needs more than plain tables. Here's what tips the balance for me:

- **It's strict.** It refuses a date like `2026-02-30`, a string in a number column and a value too long for its column. Your data stays the shape you declared.
- **It has real types.** Timestamps with time zones, arrays, JSON you can index, ranges, UUIDs.
- **It has search built in.** We'll build a search engine over my posts in one chapter, with no extra service.
- **It handles concurrency well.** Readers never wait for writers.
- **It's everywhere.** Every cloud has a managed PostgreSQL, and so do Supabase, Neon and many others.

The rest of this book shows each of these on the posts database.

Installing it

You can install PostgreSQL with a package manager, run it in a container, or use a hosted service. I'll show the first two here, and the hosting options at the end of the book.

macOS with Homebrew

On a Mac, Homebrew has one formula per major version. I install 18:

```
brew install postgresql@18
```

The installation prints some notes at the end. These are the ones that matter:

```
This formula has created a default database cluster with:
  initdb --locale=en_US.UTF-8 -E UTF-8 /opt/homebrew/var/postgresql@18
```

```
postgresql@18 is keg-only, which means it was not symlinked into /opt/homebrew,
because this is an alternate version of another formula.

If you need to have postgresql@18 first in your PATH, run:
  fish_add_path /opt/homebrew/opt/postgresql@18/bin

The following postgresql@18 executables are shadowed by linked postgresql@14 commands in /opt/homebrew/
    ↪ bin:
clusterdb
createdb
createuser
```

Two things to notice.

The formula already created a cluster in `/opt/homebrew/var/postgresql@18`, so there's no `initdb` to run.

And "keg-only" means the commands are not on your `PATH`. On my Mac that's a real problem, because I also have PostgreSQL 14 installed from years ago, and it *is* linked:

```
psql --version
```

```
psql (PostgreSQL) 14.24 (Homebrew)
```

So typing `psql` runs the old client. Put the 18 tools first in your `PATH`, in `~/.zshrc`:

```
export PATH="/opt/homebrew/opt/postgresql@18/bin:$PATH"
```

Then start the server as a background service, which also starts it at every login:

```
brew services start postgresql@18
```

```
==> Successfully started `postgresql@18` (label: sh.brew.postgresql@18)
```

```
brew services list
```

```
Name          Status  User        File
postgresql@18 started flaviocopes ~/Library/LaunchAgents/sh.brew.postgresql@18.plist
```

Homebrew created a superuser named after my macOS user, so I can connect right away, without a password:

```
psql -d postgres -c "SELECT version()"
```

```
                                                  version
-------------------------------------------------------------------------------------------------------
 PostgreSQL 18.7 (Homebrew) on aarch64-apple-darwin27.2.0, compiled by Apple clang version 21.0.0 (
    ↪ clang-2100.4.1.1), 64-bit
(1 row)
```

The server log is in `/opt/homebrew/var/log/postgresql@18.log`. When the service says `error` instead of `started`, the reason is at the end of that file.

A cluster of your own with initdb

For this book I didn't use the Homebrew cluster. I created a separate one in a test folder, on port 5418, so I could break it, restart it and throw it away without touching anything else.

`initdb` creates a data directory:

```
initdb -D pgdata
```

The first time, it failed:

```
initdb: error: invalid locale settings; check LANG and LC_* environment variables
```

The shell I was running had no locale set. PostgreSQL uses the locale to decide how text sorts, so it refuses to guess. With `LANG=en_US.UTF-8` it worked:

```
The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

creating directory pgdata ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default "max_connections" ... 100
selecting default "shared_buffers" ... 128MB
selecting default time zone ... Europe/Rome
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or --auth-local and --
    ↪ auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D pgdata -l logfile start
```

"Data page checksums are enabled" is new in PostgreSQL 18: checksums are on by default now. They let PostgreSQL detect a corrupted page on disk. Remember this line, because it caused the first failure in the upgrade chapter.

The warning about "trust" authentication is important too. It means anyone on this machine can connect as any user without a password. That's fine on a laptop, and we'll tighten it in the chapter on connections.

`pg_ctl` starts and stops the server. I pass the port and the folder for the local socket:

```
pg_ctl -D pgdata -o "-p 5418 -k /tmp/pgb" -l pg.log start
```

```
waiting for server to start.... done
server started
```

`pg_ctl -D pgdata stop` stops it, and `pg_ctl -D pgdata restart` restarts it.

Ubuntu

On a Linux server I use the packages from the PostgreSQL project (PGDG), because the version in Ubuntu's own repository is whatever was current when that Ubuntu release came out. I tested these steps in a fresh Ubuntu 24.04 container.

The `postgresql-common` package ships a script that adds the PGDG repository:

```
sudo apt install -y postgresql-common
sudo /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
sudo apt install -y postgresql-18
```

On Ubuntu the package creates a cluster called `main` and starts it. `pg_lsclusters` lists the clusters on the machine:

```
Ver Cluster Port Status Owner    Data directory              Log file
18  main    5432 online postgres /var/lib/postgresql/18/main /var/log/postgresql/postgresql-18-main.log
```

The layout is different from macOS. The data is in `/var/lib/postgresql/18/main`, but the configuration files live in `/etc/postgresql/18/main/`. And the superuser is called `postgres`, a Linux user as well as a database user.

You become that user to connect:

```
sudo -u postgres psql -c 'select version()'
```

```
                                                  version
-------------------------------------------------------------------------------------------------------
 PostgreSQL 18.7 (Ubuntu 18.7-1.pgdg24.04+5) on aarch64-unknown-linux-gnu, compiled by gcc (Ubuntu
    ↪ 13.3.0-6ubuntu2~24.04.1) 13.3.0, 64-bit
(1 row)
```

If you try as root, you get refused, once on the local socket and once over TCP:

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL:  Peer
    ↪ authentication failed for user "postgres"
psql: error: connection to server at "127.0.0.1", port 5432 failed: fe_sendauth: no password supplied
```

”Peer authentication” means PostgreSQL asks the operating system who is connecting, and only lets the Linux user `postgres` in as the database user `postgres`. Ubuntu's default `pg_hba.conf` uses it for local connections, and passwords for TCP:

```
local   all             postgres                                peer
local   all             all                                     peer
host    all             all             127.0.0.1/32            scram-sha-256
host    all             all             ::1/128                 scram-sha-256
```

Debian and Ubuntu also add their own tools to manage several clusters: `pg_createcluster`, `pg_ctlcluster`, `pg_dropcluster`. Creating a second cluster on another port takes one command:

```
sudo pg_createcluster 18 second --port 5433
```

```
Ver Cluster Port Status Owner    Data directory               Log file
18  main    5432 online postgres /var/lib/postgresql/18/main   /var/log/postgresql/postgresql-18-main.
    ↪ log
18  second  5433 down   postgres /var/lib/postgresql/18/second /var/log/postgresql/postgresql-18-second
    ↪ .log
```

For running PostgreSQL on a server for real, with a firewall, backups to another machine and monitoring, my Linux Server Handbook covers the whole setup.

Docker

The official image is the fastest way to get a throwaway server:

```
docker run --name pg-handbook -e POSTGRES_PASSWORD=secret -p 5432:5432 -d postgres:18
```

Mount a volume on the data directory if you want the data to survive the container. Be careful with the path: from PostgreSQL 18 the image keeps its data in `/var/lib/postgresql/18/docker`, and the volume goes on `/var/lib/postgresql`, one level higher than in older images.

Loading the posts database

With the server running, I create the database and load my posts. `createdb` is a small wrapper around `CREATE DATABASE`:

```
createdb blog
```

The script that builds the data reads every published Markdown file in my blog's repository and prints SQL. It uses the `yaml` package from npm to read the front-matter:

```
// node load-posts.mjs ~/www/flaviocopes.com/src/posts > posts.sql
import { readdir, readFile } from 'node:fs/promises'
import path from 'node:path'
import { parse } from 'yaml'

const dir = process.argv[2] || 'src/posts'
const today = new Date()
const quote = (value) => `'${String(value).replaceAll("'", "''")}'`

const posts = []
for (const file of (await readdir(dir)).sort()) {
  if (!file.endsWith('.md')) continue
  const source = await readFile(path.join(dir, file), 'utf8')
  const match = source.match(/^---\r?\n([\s\S]*?)\r?\n---\r?\n([\s\S]*)$/)
  if (!match) continue
  const data = parse(match[1]) ?? {}
  const body = match[2]
```

```
  const date = new Date(data.date)
  if (data.draft || Number.isNaN(date.getTime()) || date > today) continue
  const tags = [data.tag ?? []].flat().map((tag) => String(tag).toLowerCase())
  const words = body.replace(/```[\s\S]*?```/g, '').split(/\s+/).filter(Boolean).length
  posts.push({
    slug: file.replace(/\.md$/, ''),
    title: String(data.title),
    date: date.toISOString().slice(0, 10),
    words,
    body,
    frontmatter: JSON.stringify(data),
    tags: [...new Set(tags)],
  })
}

posts.sort((a, b) => a.date.localeCompare(b.date) || a.slug.localeCompare(b.slug))
const tagNames = [...new Set(posts.flatMap((post) => post.tags))].sort()

console.log(`BEGIN;

CREATE TABLE posts (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  slug text NOT NULL UNIQUE,
  title text NOT NULL,
  published_on date NOT NULL,
  words integer NOT NULL,
  body text NOT NULL,
  frontmatter jsonb NOT NULL
);

CREATE TABLE tags (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  name text NOT NULL UNIQUE
);

CREATE TABLE post_tags (
  post_id bigint NOT NULL REFERENCES posts (id) ON DELETE CASCADE,
  tag_id bigint NOT NULL REFERENCES tags (id) ON DELETE CASCADE,
  PRIMARY KEY (post_id, tag_id)
);
`)

for (const post of posts) {
  console.log(
    `INSERT INTO posts (slug, title, published_on, words, body, frontmatter) VALUES (${quote(post.slug)
    ↪ }, ${quote(post.title)}, ${quote(post.date)}, ${post.words}, ${quote(post.body)}, ${quote(post.
    ↪ frontmatter)});`,
  )
}
for (const name of tagNames) {
  console.log(`INSERT INTO tags (name) VALUES (${quote(name)});`)
}
for (const post of posts) {
  for (const tag of post.tags) {
    console.log(
      `INSERT INTO post_tags (post_id, tag_id) SELECT p.id, t.id FROM posts p, tags t WHERE p.slug = ${
    ↪ quote(post.slug)} AND t.name = ${quote(tag)};`,
    )
  }
}
console.log('\nCOMMIT;')
```

The output is a 17 MB SQL file. The whole thing is one transaction, between `BEGIN` and `COMMIT`, so if anything fails nothing gets loaded:

```
node load-posts.mjs ~/www/flaviocopes.com/src/posts > posts.sql
psql -q -d blog -f posts.sql
```

It took 0.72 seconds. Then a quick check:

```
SELECT count(*), sum(words) FROM posts;
```

```
 count |   sum
-------+---------
  2193 | 2088507
(1 row)
```

2,193 posts and 2,088,507 words, with 49 tags and 2,239 links between posts and tags. The same numbers as in the SQL Handbook, plus the text.

psql

`psql` is the command-line client that comes with PostgreSQL. There are graphical clients too, but `psql` is the one that's always there, on every server, and it knows things about PostgreSQL that generic tools don't.

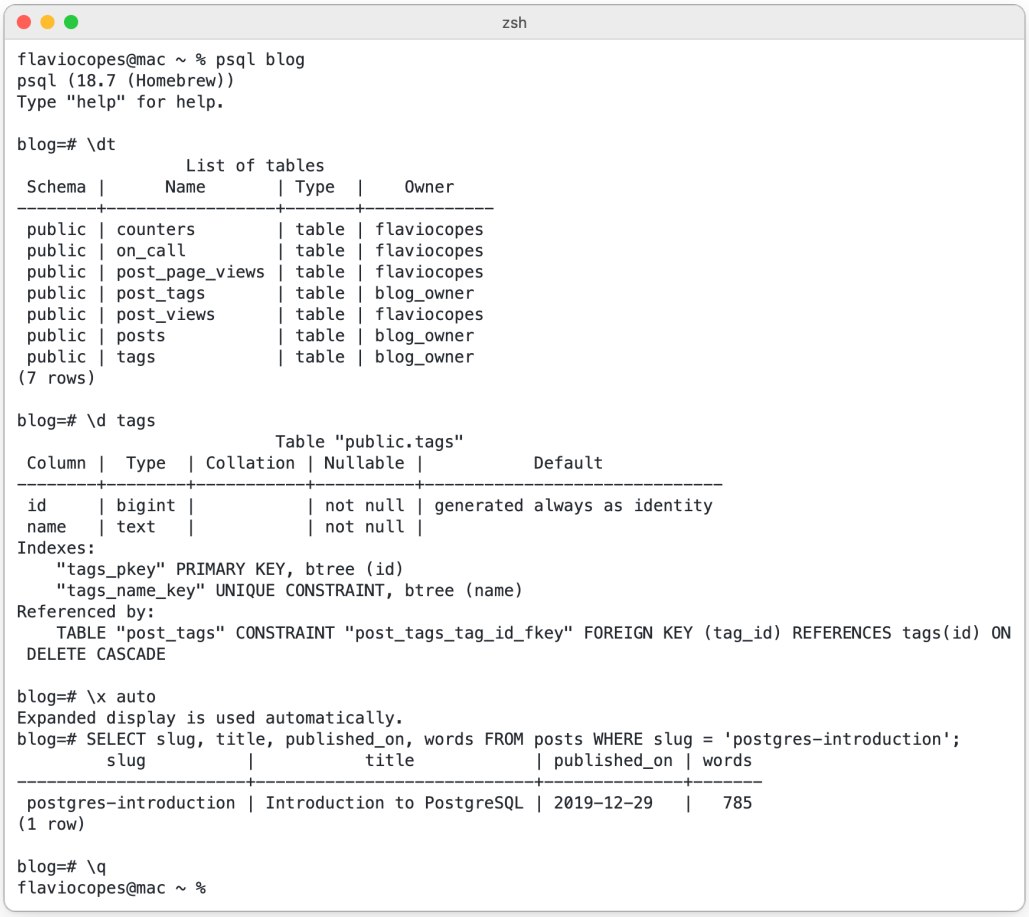
You connect by giving it a database name:

```
psql blog
```

Without options, `psql` connects through the local socket, as a database user with the same name as your system user. My test server is on a different port and socket folder, so I set two environment variables once and forget about them:

```
export PGHOST=/tmp/pgb PGPORT=5418
```

We'll see all the ways to tell `psql` where to connect in the next chapter.



Meta-commands

Inside `psql`, anything that starts with a backslash is a **meta-command**. `psql` handles it itself, often by running queries on the system catalogs for you. Anything else is SQL, sent to the server when you type `;`.

`\l` lists the databases in the cluster:

```
                                                    List of databases
   Name    |    Owner    | Encoding | Locale Provider |   Collate   |    Ctype    | Locale | ICU Rules
    ↪ |      Access privileges
-----------+-------------+----------+-----------------+-------------+-------------+--------+-----------
 blog      | flaviocopes | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |
    ↪ |
 postgres  | flaviocopes | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |
    ↪ |
 template0 | flaviocopes | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |
    ↪ | =c/flaviocopes             +
          |             |          |                 |             |             |        |
    ↪ | flaviocopes=CTc/flaviocopes
 template1 | flaviocopes | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |
    ↪ | =c/flaviocopes             +
          |             |          |                 |             |             |        |
    ↪ | flaviocopes=CTc/flaviocopes
(4 rows)
```

`\c postgres` switches to another database. `\dt` lists the tables of the current one:

```
              List of tables
```

```
 Schema |   Name    | Type  |    Owner
--------+-----------+-------+-------------
 public | post_tags | table | flaviocopes
 public | posts     | table | flaviocopes
 public | tags      | table | flaviocopes
(3 rows)
```

`\d posts` describes a table, with its columns, indexes and the foreign keys that point to it:

```
                          Table "public.posts"
    Column    |  Type   | Collation | Nullable |           Default
--------------+---------+-----------+----------+-----------------------------
 id           | bigint  |           | not null | generated always as identity
 slug         | text    |           | not null |
 title        | text    |           | not null |
 published_on | date    |           | not null |
 words        | integer |           | not null |
 body         | text    |           | not null |
 frontmatter  | jsonb   |           | not null |
Indexes:
    "posts_pkey" PRIMARY KEY, btree (id)
    "posts_slug_key" UNIQUE CONSTRAINT, btree (slug)
Referenced by:
    TABLE "post_tags" CONSTRAINT "post_tags_post_id_fkey" FOREIGN KEY (post_id) REFERENCES posts(id) ON
    ↪  DELETE CASCADE
```

Add a `+` to most commands for more detail. `\d+ tags` also shows the storage of each column, and in PostgreSQL 18 the not-null constraints by name:

```
Not-null constraints:
    "tags_id_not_null" NOT NULL "id"
    "tags_name_not_null" NOT NULL "name"
Access method: heap
```

These are the ones I use every day:

Command	What it does
`\l`	list databases
`\c name`	connect to another database
`\dt`	list tables (`\dt stats.*` for another schema)
`\d name`	describe a table, view, index or sequence
`\dn`	list schemas
`\du`	list roles
`\dp name`	show who has which privileges on a table
`\df`	list functions
`\x`	toggle expanded output (`\x auto` decides by width)
`\timing`	show how long each query takes
`\e`	edit the last query in your editor
`\i file.sql`	run a file
`\conninfo`	show what you're connected to

Command	What it does
`\?`	list all meta-commands (163 lines of them)
`\q`	quit

Expanded output

Rows with many columns, or long values, wrap into an unreadable mess. `\x` turns each row into a vertical list of `column | value` pairs:

```
\x
SELECT id, slug, title, published_on, words, frontmatter
FROM posts WHERE slug = 'postgres-introduction';
```

```
-[ RECORD 1 ]+------------------------------------------------------------
id           | 732
slug         | postgres-introduction
title        | Introduction to PostgreSQL
published_on | 2019-12-29
words        | 785
frontmatter  | {"tag": "database", "date": "2019-12-29T07:00:00+02:00", "title": "Introduction to
    ↪ PostgreSQL", "updated": "2026-11-28", "description": "Learn PostgreSQL from basics to production
    ↪ : psql commands, roles and least privilege, constraints, transactions, EXPLAIN ANALYZE indexing,
    ↪  connection pools, and pg_dump backups."}
```

I put `\x auto` in my `~/.psqlrc`, so `psql` switches by itself when a result is too wide for the terminal.

Timing

`\timing on` prints how long each statement took, measured by `psql`, so it includes the round trip to the server:

```
\timing on
SELECT count(*) FROM posts WHERE body ILIKE '%vacuum%';
```

```
 count
-------
     6
(1 row)

Time: 69.470 ms
```

70 milliseconds to read 2 million words, looking for one. Remember the number. In the full-text search chapter, an indexed search over the same text takes about 1.5 milliseconds.

Where am I?

`\conninfo` tells you what you're connected to. It's the first thing I run when something behaves strangely, because half of the time I'm on the wrong database or the

wrong server:

```
       Connection Information
      Parameter       |    Value
----------------------+-------------
 Database             | blog
 Client User          | flaviocopes
 Socket Directory     | /tmp/pgb
 Server Port          | 5418
 Options              |
 Protocol Version     | 3.0
 Password Used        | false
 GSSAPI Authenticated | false
 Backend PID          | 90010
 SSL Connection       | false
 Superuser            | on
 Hot Standby          | off
(12 rows)
```

`Backend PID` is the server process that serves this connection, the one the postmaster started for me. We'll use PIDs to find blocked queries in the locks chapter.

Running SQL from the shell

`-c` runs one command and exits, and `-f` runs a file. Both are what you use in scripts:

```
psql blog -c "SELECT count(*) FROM tags"
psql blog -f migration.sql
```

By default `psql` keeps going after an error in a file. In scripts I always add `-v ON_ERROR_STOP=1`, so the first error stops the run and `psql` exits with a non-zero code.

Another option I use in scripts is `-X`, which skips `~/.psqlrc`. Your personal settings shouldn't change what a script does.

~/.psqlrc

`psql` runs this file at startup. Here's a short one, the one I used while writing this book:

```
\set QUIET 1
\x auto
\timing on
\pset null ''
\set HISTCONTROL ignoredups
\set COMP_KEYWORD_CASE upper
\unset QUIET
```

`\pset null ' '` is the most useful line. By default `NULL` shows as an empty cell, exactly like an empty string, and you can't tell them apart.

Connecting, and "could not connect"

Every client, `psql` or your application, needs the same five things to connect: a host, a port, a database, a user and, usually, a password.

Connection strings

Applications usually get all five in one URL:

```
postgresql://blog_app:app-secret-123@127.0.0.1:5418/blog
```

The parts are `postgresql://user:password@host:port/database`. Options go after a `?`, like `?sslmode=require` for an encrypted connection, which you want for any database on another machine.

If the password contains characters like `@`, `:` or `/`, you need to percent-encode them in the URL (`@` becomes `%40`). It's the most common reason a connection string that "looks right" fails.

`psql` accepts the same URL instead of a database name:

```
psql "postgresql://blog_app:app-secret-123@127.0.0.1:5418/blog"
```

Environment variables

Every PostgreSQL client built on `libpq`, which includes `psql`, `pg_dump` and many drivers, reads these variables for the parts you didn't specify:

Variable	Meaning	Default
`PGHOST`	host name, IP, or socket folder	the local socket
`PGPORT`	port	5432
`PGDATABASE`	database	same as the user
`PGUSER`	user	your system user
`PGPASSWORD`	password	none

`PGPASSWORD` works, but it's visible to other processes on some systems. For your own machine, a `~/.pgpass` file is better. Each line is `host:port:database:user:password`:

```
127.0.0.1:5418:blog:blog_app:app-secret-123
```

The file must be readable only by you (`chmod 600 ~/.pgpass`), otherwise `psql` ignores it.

Reading connection errors

When a connection fails, the error message tells you how far it got. I made each one happen on my test server.

Nothing is listening. On the local socket, the socket file isn't there:

```
psql: error: connection to server on socket "/tmp/pgb/.s.PGSQL.5419" failed: No such file or directory
    Is the server running locally and accepting connections on that socket?
```

Over TCP, the operating system refuses the connection:

```
psql: error: connection to server at "127.0.0.1", port 5419 failed: Connection refused
    Is the server running on that host and accepting TCP/IP connections?
```

Both mean the same thing: the server isn't running, or it's on another port (here I typed 5419 instead of 5418), or it doesn't listen on that address. By default PostgreSQL only listens on `localhost`:

```
SHOW listen_addresses;
```

```
 listen_addresses
------------------
 localhost
(1 row)
```

To accept connections from other machines, set `listen_addresses = '*'` in `postgresql.conf`, restart, and open the port in the firewall. Then `pg_hba.conf` decides who actually gets in.

The server answered, and said no. These errors start with `FATAL`, which means you reached PostgreSQL:

```
psql: error: connection to server on socket "/tmp/pgb/.s.PGSQL.5418" failed: FATAL:  database "blgo"
    ↪ does not exist
psql: error: connection to server on socket "/tmp/pgb/.s.PGSQL.5418" failed: FATAL:  role "app" does
    ↪ not exist
psql: error: connection to server at "127.0.0.1", port 5418 failed: FATAL:  password authentication
    ↪ failed for user "blog_app"
```

A typo in the database name, a user that doesn't exist, a wrong password. Note the second one: PostgreSQL says **role**, not user. The next part of the book explains why.

pg_hba.conf

`pg_hba.conf` is the server's front door. Each line says: for this type of connection, to this database, as this user, from this address, use this authentication method. PostgreSQL reads the lines top to bottom and uses the first one that matches.

After `initdb`, my test cluster trusted everyone. I changed the TCP lines to require a password:

```
local   all             all                                     trust
host    all             all             127.0.0.1/32            scram-sha-256
host    all             all             ::1/128                 scram-sha-256
local   replication     all                                     trust
```

```
host    replication     all             127.0.0.1/32            scram-sha-256
host    replication     all             ::1/128                 scram-sha-256
```

The methods you'll meet:

Method	What it checks
`trust`	nothing, everyone gets in
`peer`	the operating system user matches the database user (local socket only)
`scram-sha-256`	a password, without ever sending it over the wire
`md5`	a password, with an old and weaker hash (avoid it)
`reject`	nobody gets in

Changes to `pg_hba.conf` need a reload, not a restart:

```
pg_ctl -D pgdata reload
```

```
server signaled
```

Or, from `psql`, `SELECT pg_reload_conf();`.

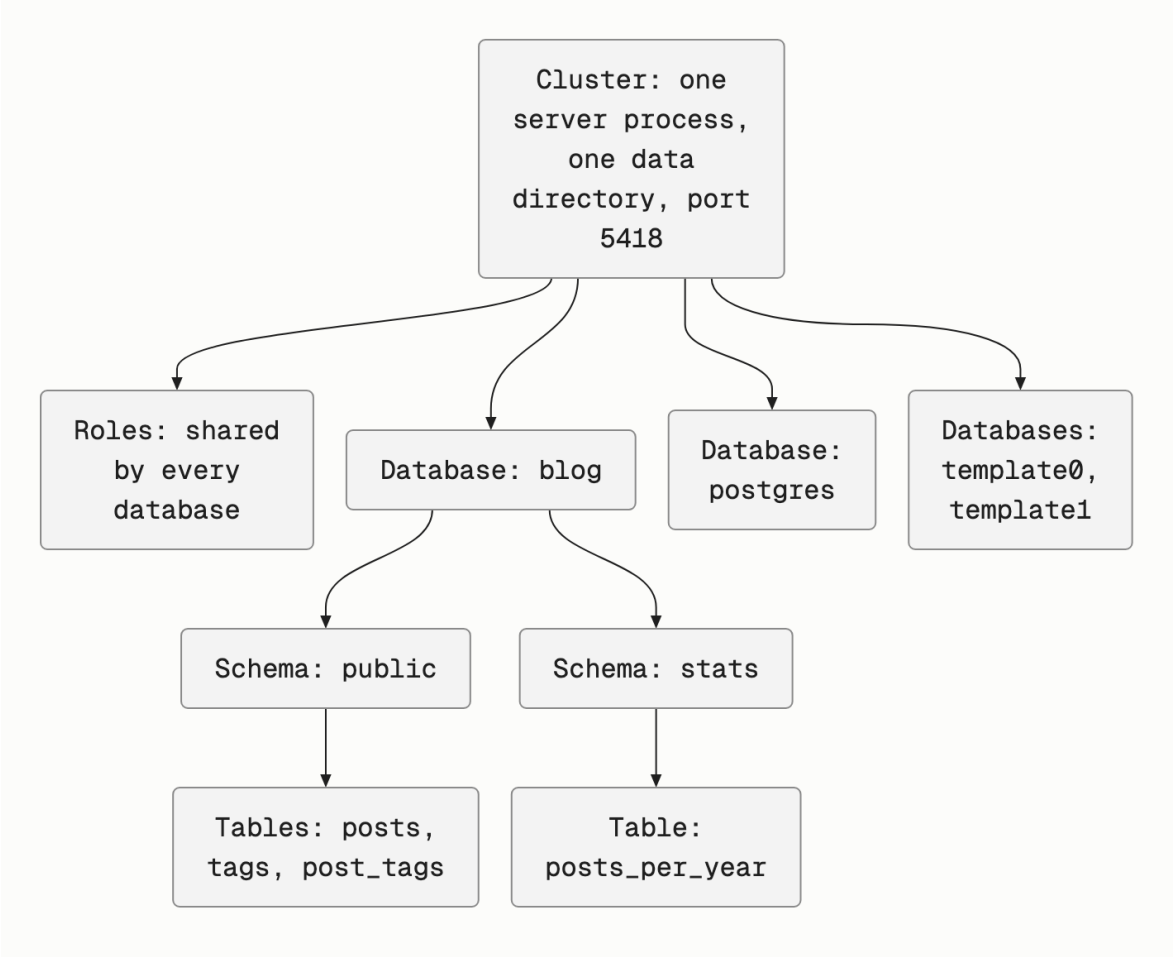
When no line matches at all, the error says so. I removed the `::1` line and connected over IPv6:

```
psql: error: connection to server at "::1", port 5418 failed: FATAL:  no pg_hba.conf entry for host
    ↪ "::1", user "blog_app", database "blog", no encryption
```

Add a line for that address, or connect from somewhere that's allowed.

Clusters, databases and schemas

We've seen that a cluster holds databases. Inside a database there's one more level: **schemas**.



So the full name of my posts table is `blog.public.posts`: database, schema, table.

A connection sees one database

When you connect, you pick a database, and that connection can only see that database. A query that reaches into another one fails:

```
SELECT count(*) FROM postgres.public.pg_class;
```

```
ERROR:  cross-database references are not implemented: "postgres.public.pg_class"
LINE 1: SELECT count(*) FROM postgres.public.pg_class
                             ^
```

This is a real difference from MySQL, where ”database” and ”schema” mean the same thing and one query can join tables from two of them. In PostgreSQL, if two sets of tables need to be joined, they belong in the same database, maybe in different schemas.

Roles are the exception. Roles belong to the cluster, so the same user can log into every database (if it's allowed to).

The public schema

Every new database has a schema called `public`, and when you don't say which schema you mean, that's usually where things go. All the tables I loaded are in `public`.

`\dn+` lists the schemas with their permissions:

```
                                  List of schemas
  Name  |       Owner       |           Access privileges            |      Description
--------+-------------------+----------------------------------------+-------------------------
 public | pg_database_owner | pg_database_owner=UC/pg_database_owner+| standard public schema
        |                   | =U/pg_database_owner                   |
(1 row)
```

Read the privileges like this: the owner of the database can use (`U`) and create in (`C`) the schema, and everyone else (the empty name before `=`) can only use it. Before PostgreSQL 15, everyone could also create tables in `public`. Many old tutorials assume they still can, and that's the source of a common error we'll see in the next chapter.

Making a second schema

I use schemas to keep things apart inside one database. Let's put some reports in their own schema:

```
CREATE SCHEMA stats;

CREATE TABLE stats.posts_per_year AS
  SELECT extract(year FROM published_on)::int AS year, count(*) AS posts
  FROM posts GROUP BY 1;
```

```
CREATE SCHEMA
SELECT 13
```

`CREATE TABLE ... AS` creates the table from a query, and copies the result into it. 13 rows, one for each year I published something, from 2012 to 2026 (there's nothing in 2015 and 2016).

search_path, and "relation does not exist"

Now let's read the new table without saying which schema it's in:

```
SELECT * FROM posts_per_year ORDER BY year DESC LIMIT 3;
```

```
ERROR:  relation "posts_per_year" does not exist
LINE 1: SELECT * FROM posts_per_year ORDER BY year DESC LIMIT 3
                      ^
```

The table exists. PostgreSQL just didn't look for it in `stats`. When a name has no schema, PostgreSQL looks in the schemas listed in the `search_path` setting, in order:

```
SHOW search_path;
```

```
   search_path
-----------------
 "$user", public
(1 row)
```

First a schema with the same name as the current user (I have none), then `public`. `stats` isn't on the list.

There are two fixes. Use the schema name:

```
SELECT * FROM stats.posts_per_year ORDER BY year DESC LIMIT 3;
```

```
 year | posts
------+-------
 2026 |   509
 2025 |    25
 2024 |    66
(3 rows)
```

Or add the schema to the search path, for this session:

```
SET search_path = public, stats;
```

You can also make it permanent for a user (`ALTER ROLE ... SET search_path = ...`) or for a database (`ALTER DATABASE ... SET ...`).

In application code I prefer the first fix. A query with the schema in it works the same no matter how the connection was set up.

The case of the missing capital letter

"relation does not exist" has a second common cause, and it has nothing to do with schemas. PostgreSQL folds names to lowercase unless you put them in double quotes:

```
CREATE TABLE "Drafts" (id int);
SELECT * FROM Drafts;
```

```
CREATE TABLE
ERROR:  relation "drafts" does not exist
LINE 1: SELECT * FROM Drafts
                      ^
```

The table is called `Drafts`, with a capital D, because the name was quoted. The query asked for `Drafts` without quotes, which PostgreSQL read as `drafts`. Only `SELECT count(*) FROM "Drafts"` finds it.

ORMs and GUI tools sometimes create quoted mixed-case names. My advice is to use lowercase names with underscores everywhere, and never think about quotes again.

Who am I, where am I

These functions answer the questions you ask when an error makes no sense:

```
SELECT current_database(), current_schema, current_user, session_user;
```

```
 current_database | current_schema | current_user | session_user
------------------+----------------+--------------+--------------
 blog             | public         | flaviocopes  | flaviocopes
(1 row)
```

`current_schema` is the first schema in the search path that exists, so it's where a `CREATE TABLE` without a schema would put the table.

Roles and permissions

So far I've been connecting as `flaviocopes`, the superuser `initdb` created. A superuser skips every permission check. That's fine for setting things up, and wrong for an application.

If the application connects as a superuser, a bug in it (or an SQL injection) can do anything: drop every table, read every database, even run programs on the server. The fix is to give each program its own role, with only the permissions it needs.

Users and groups are both roles

PostgreSQL has one concept for both users and groups: the **role**.

A role with the `LOGIN` attribute can connect, so it works like a user. A role without it can't connect, but it can own objects and be granted to other roles, so it works like a group. `CREATE USER` is the same as `CREATE ROLE ... LOGIN`.

Three roles for the blog

Here's the setup I use for an application:

- `blog_owner` owns the tables. Nobody logs in as it. Migrations run with its permissions.
- `blog_app` is what the application connects as. It can read and write rows, and nothing else.
- `blog_reader` is for reports and dashboards. It can only read.

```
CREATE ROLE blog_owner NOLOGIN;
ALTER DATABASE blog OWNER TO blog_owner;
ALTER TABLE posts OWNER TO blog_owner;
ALTER TABLE tags OWNER TO blog_owner;
ALTER TABLE post_tags OWNER TO blog_owner;
ALTER SCHEMA stats OWNER TO blog_owner;
ALTER TABLE stats.posts_per_year OWNER TO blog_owner;

CREATE ROLE blog_app LOGIN PASSWORD 'app-secret-123';
CREATE ROLE blog_reader LOGIN PASSWORD 'reader-secret-123';
```

`\du` shows them:

```
                             List of roles
  Role name  |                         Attributes
-------------+------------------------------------------------------------
 blog_app    |
 blog_owner  | Cannot login
 blog_reader |
 flaviocopes | Superuser, Create role, Create DB, Replication, Bypass RLS
```

The passwords are never stored. PostgreSQL keeps a salted SCRAM hash:

```
SELECT rolname, left(rolpassword, 30) FROM pg_authid WHERE rolname LIKE 'blog%';
```

```
   rolname   |              left
-------------+--------------------------------
 blog_owner  |
 blog_app    | SCRAM-SHA-256$4096:/SDuZZPPlLg
 blog_reader | SCRAM-SHA-256$4096:XE46ypnVBdQ
(3 rows)
```

A real password should be long and random, and come from a secret manager or an environment variable, not from your shell history. In `psql`, `\password blog_app` asks for it without echoing it, and sends only the hash to the server.

A new role can do almost nothing

Let's connect as `blog_app` and read the posts:

```
PGPASSWORD=app-secret-123 psql -h 127.0.0.1 -U blog_app -d blog -c "select count(*) from posts"
```

```
ERROR:  permission denied for table posts
```

It could connect, though. Why? Because two default grants go to `PUBLIC`, a pseudo-role that means every role:

```
SELECT has_database_privilege('blog_app', 'blog', 'CONNECT');
SELECT has_schema_privilege('blog_app', 'public', 'USAGE'),
       has_schema_privilege('blog_app', 'public', 'CREATE');
```

```
 has_database_privilege
------------------------
 t
(1 row)

 has_schema_privilege | has_schema_privilege
----------------------+----------------------
 t                    | f
(1 row)
```

Every role can connect to every database and look up names in `public`. Nobody but the owner can create tables there, and nobody but the owner can read the tables.

Granting what each role needs

Permissions work in layers. To read a table a role needs `CONNECT` on the database, `USAGE` on the schema, and `SELECT` on the table. I start by taking away the default `CONNECT` and giving it only to my two roles:

```
REVOKE CONNECT ON DATABASE blog FROM PUBLIC;
GRANT CONNECT ON DATABASE blog TO blog_app, blog_reader;
GRANT USAGE ON SCHEMA public TO blog_app, blog_reader;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO blog_reader;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO blog_app;
```

Now each role can do its job, and no more. The app can read:

```
 count
-------
  2193
(1 row)
```

The app can add a tag:

```
INSERT INTO tags (name) VALUES ('postgresql') RETURNING id, name;
```

```
 id |    name
----+------------
 50 | postgresql
(1 row)
```

But it can't change the schema:

```
CREATE TABLE hack (id int);
```

```
ERROR:  permission denied for schema public
LINE 1: CREATE TABLE hack (id int)
                    ^
```

And the reader can't write:

```
DELETE FROM tags WHERE name = 'postgresql';
```

```
ERROR:  permission denied for table tags
```

Note that the app could insert into `tags` without any permission on the sequence behind its `id` column. Identity columns don't need one. Old `serial` columns do, and that's another error you'll meet in older tutorials: `permission denied for sequence tags_id_seq`.

`\dp tags` shows the result in PostgreSQL's compact notation:

```
                                Access privileges
 Schema | Name | Type  |       Access privileges        | Column privileges | Policies
--------+------+-------+--------------------------------+-------------------+----------
 public | tags | table | blog_owner=arwdDxtm/blog_owner+|                   |
        |      |       | blog_reader=r/blog_owner      +|                   |
        |      |       | blog_app=arwd/blog_owner       |                   |
(1 row)
```

Each letter is one privilege: `r` read (`SELECT`), `a` append (`INSERT`), `w` write (`UPDATE`), `d` delete, and the owner also has `D` (`TRUNCATE`), `x` (`REFERENCES`), `t` (`TRIGGER`) and `m` (`MAINTAIN`, new in PostgreSQL 17). After the slash is the role that granted it.

One thing surprised me while testing this: `blog_reader` can still connect to the `postgres` database. I revoked `CONNECT` on `blog` only, and the default grant to `PUBLIC` is still there on the others. If that matters, revoke it on each database.

Tables created later

`GRANT ... ON ALL TABLES` covers the tables that exist now. A table created tomorrow starts with no grants, and the reader can't see it:

```
CREATE TABLE subscribers_tmp (id int);
ALTER TABLE subscribers_tmp OWNER TO blog_owner;
```

```
ERROR:  permission denied for table subscribers_tmp
```

Default privileges fix that. They're grants that PostgreSQL applies automatically to every new object a given role creates:

```
ALTER DEFAULT PRIVILEGES FOR ROLE blog_owner IN SCHEMA public
  GRANT SELECT ON TABLES TO blog_reader;
ALTER DEFAULT PRIVILEGES FOR ROLE blog_owner IN SCHEMA public
  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO blog_app;
ALTER DEFAULT PRIVILEGES FOR ROLE blog_owner IN SCHEMA public
  GRANT USAGE ON SEQUENCES TO blog_app;
```

The `FOR ROLE blog_owner` part matters. Default privileges apply to objects created *by that role*. If a migration runs as someone else, its tables don't get them. So migrations should create tables as `blog_owner`:

```
SET ROLE blog_owner;
CREATE TABLE subscribers_tmp2 (id int);
RESET ROLE;
```

And this time the reader can read the new table right away. `\ddp` lists the default privileges:

```
                  Default access privileges
   Owner    | Schema |   Type   |    Access privileges
------------+--------+----------+--------------------------
 blog_owner | public | sequence | blog_app=U/blog_owner
 blog_owner | public | table    | blog_app=arwd/blog_owner+
            |        |          | blog_reader=r/blog_owner
(2 rows)
```

Limits per role

Roles can also carry settings. Two I set on every application role:

```
ALTER ROLE blog_app CONNECTION LIMIT 20;
ALTER ROLE blog_app SET statement_timeout = '5s';
```

The connection limit stops a misconfigured app from taking all the server's connections. The statement timeout cancels any query that runs longer than 5 seconds, so one runaway query can't hold locks forever. When we build the Node.js app later, we'll see this timeout fire.

Roles compared

Role	Can log in	Owns tables	Can read	Can write rows	Can change schema
`flaviocopes` (superuser)	yes	no	everything	everything	everything
`blog_owner`	no	yes	yes	yes	yes
`blog_app`	yes	no	yes	yes	no
`blog_reader`	yes	no	yes	no	no

Types worth knowing

The SQL Handbook covered the types every database has. PostgreSQL has many more, and picking the right one saves you code. These are the ones I reach for.

text, varchar and char

In PostgreSQL, `text` and `varchar` are the same type underneath, with the same performance. `varchar(n)` adds a length check:

```
CREATE TABLE t_types (a varchar(5), b text, c char(5));
INSERT INTO t_types VALUES ('hello', 'hello', 'hi');
INSERT INTO t_types (a) VALUES ('hello!');
SELECT a, b, c, length(c), octet_length(c), c = 'hi' AS eq FROM t_types;
```

```
CREATE TABLE
INSERT 0 1
ERROR:  value too long for type character varying(5)
   a   |   b   |   c   | length | octet_length | eq
-------+-------+-------+--------+--------------+----
 hello | hello | hi    |      2 |            5 | t
(1 row)
```

`char(5)` pads the value with spaces to 5 characters and stores all 5 (`octet_length` is 5), then ignores the padding when comparing. It's a leftover from older systems. I never use it.

My default is `text`. When there's a real limit, like a country code, I prefer a `CHECK` constraint, because changing a check later is cheaper than changing a column type (the migrations chapter shows by how much).

timestamptz, and the time zone trap

PostgreSQL has two timestamp types, and the names are misleading:

- `timestamp` (without time zone) stores a date and a time, and nothing else
- `timestamptz` (with time zone) stores an exact moment, in UTC, and shows it in the session's time zone

Let's see the difference. My session is in Rome:

```
SET timezone = 'Europe/Rome';
SELECT '2026-11-28 19:00'::timestamptz AS rome, '2026-11-28 19:00'::timestamp AS naive;
```

```
          rome          |        naive
------------------------+---------------------
 2026-11-28 19:00:00+01 | 2026-11-28 19:00:00
(1 row)
```

The same moment, read by someone in New York:

```
SET timezone = 'America/New_York';
SELECT '2026-11-28 19:00+01'::timestamptz AS shown_in_ny;
```

```
      shown_in_ny
------------------------
```

```
 2026-11-28 13:00:00-05
(1 row)
```

19:00 in Rome is 13:00 in New York, and `timestamptz` knows it.

Now the trap. I insert `now()` into both kinds of column from a session in Rome, then read them back from a session in UTC, like a server would:

```
CREATE TEMP TABLE ev (naive timestamp, aware timestamptz);
SET timezone = 'Europe/Rome';
INSERT INTO ev VALUES (now(), now());
SET timezone = 'UTC';
SELECT naive, aware FROM ev;
```

```
           naive            |              aware
----------------------------+-------------------------------
 2026-11-28 19:12:47.393182 | 2026-11-28 18:12:47.393182+00
(1 row)
```

The `timestamp` column says 19:12, which in UTC is an hour in the future. It stored the wall-clock time of whoever wrote it and threw away where that was. The `timestamptz` column is right.

Use `timestamptz` for anything that happened at a moment: created at, published at, viewed at. Use `date` for calendar days, like `published_on` in my posts table.

numeric for money

`float8` (double precision) is fast and approximate. `numeric` is exact:

```
SELECT 0.1::float8 + 0.2::float8 = 0.3 AS float_eq,
       0.1::numeric + 0.2::numeric = 0.3 AS numeric_eq,
       0.1::float8 + 0.2::float8 AS float_sum;
```

```
 float_eq | numeric_eq |      float_sum
----------+------------+---------------------
 f        | t          | 0.30000000000000004
(1 row)
```

Money goes in `numeric(12,2)`, or in an integer number of cents. Never in `float`. And note that `integer` tops out at about 2.1 billion:

```
SELECT 2147483647::int + 1;
```

```
ERROR:  integer out of range
```

That's why my ID columns are `bigint`. A busy table can pass 2 billion rows, and changing the type of a primary key on a big table is painful.

boolean

`boolean` accepts many spellings and stores one bit of meaning:

```
SELECT 'yes'::boolean, 'off'::boolean, 't'::boolean, '1'::boolean;
```

```
 bool | bool | bool | bool
------+------+------+------
 t    | f    | t    | t
(1 row)
```

Arrays

Any type can be an array. I don't often store arrays in tables (a separate table is usually better, like my `post_tags`), but they're great in query results. `array_agg` collects values into one:

```
SELECT p.slug, array_agg(t.name ORDER BY t.name) AS tags
FROM posts p
JOIN post_tags pt ON pt.post_id = p.id
JOIN tags t ON t.id = pt.tag_id
GROUP BY p.slug
HAVING count(*) > 2
ORDER BY p.slug
LIMIT 5;
```

```
               slug                |            tags
-----------------------------------+----------------------------
 fix-error-package-json-prettier   | {astro,tailwind,tools}
 fixing-ts-issues-in-vs-code-astro | {astro,tools,typescript}
 htmx-trigger-request-via-js-event | {alpine,htmx,js}
 run-pocketbase-on-flyio           | {database,docker,services}
 the-aha-stack                     | {alpine,astro,htmx}
(5 rows)
```

And there are operators to work with them:

```
SELECT 'ai' = ANY (array['ai', 'node']) AS has_ai,
       array['ai', 'node'] @> array['node'] AS contains,
       cardinality(array['a', 'b', 'c']);
```

```
 has_ai | contains | cardinality
--------+----------+-------------
 t      | t        |           3
(1 row)
```

`= ANY` is also how you pass a list to a query from application code: `WHERE id = ANY($1)` with an array as the one parameter.

uuid, and uuidv7 in PostgreSQL 18

UUIDs are IDs you can generate anywhere, without asking the database for the next number. PostgreSQL 18 can generate both common versions:

```
SELECT uuidv4() AS v4, uuidv7() AS v7;
```

```
                  v4                  |                  v7
--------------------------------------+--------------------------------------
 28f8b248-a472-424b-9edf-763917875fbf | 01a2235c-16f7-7478-a6bc-99cce5b7f1c3
(1 row)
```

A v4 is fully random. A v7 starts with a timestamp, so values created one after another sort together:

```
SELECT uuidv7() FROM generate_series(1, 3);
```

```
                uuidv7
--------------------------------------
 01a2235c-16f7-763b-a53a-e3f5c68a6759
 01a2235c-16f7-763f-a966-07a1720fac1e
 01a2235c-16f7-7643-8d66-9382bad2aa2f
(3 rows)
```

That matters for primary keys. New rows land at the end of the index instead of at random places in it, which keeps inserts fast on big tables. And you can read the time back:

```
SELECT uuid_extract_timestamp(uuidv7());
```

```
   uuid_extract_timestamp
----------------------------
 2026-11-28 19:12:47.492+01
(1 row)
```

If you need UUID keys, use v7. Before PostgreSQL 18 you had to generate them in the application or with an extension.

Ranges

A range is a start and an end in one value. `daterange`, `tstzrange`, `int4range` and others come built in, with operators like `@>` (contains) and `<@` (is contained by):

```
SELECT daterange('2026-01-01', '2027-01-01') @> '2026-11-28'::date AS in_2026,
       count(*) FILTER (WHERE published_on <@ daterange('2026-01-01', '2027-01-01')) AS posts_2026
FROM posts;
```

```
 in_2026 | posts_2026
---------+------------
 t       |        509
(1 row)
```

By default the start is included and the end isn't, written `[2026-01-01,2027-01-01)`. We'll use a range for something more interesting in the next chapter: making two bookings for the same room impossible.

Dates, intervals and generate_series

Subtracting two dates gives a number of days, and `age()` gives a readable interval:

```
SELECT max(published_on) - min(published_on) AS days,
       age(max(published_on), min(published_on))
FROM posts;
```

```
 days |           age
------+-------------------------
 5248 | 14 years 4 mons 12 days
(1 row)
```

`generate_series` creates rows out of nothing. With dates, it's how you get a row for every day, including the days with no posts:

```
SELECT d::date, count(p.id)
FROM generate_series('2026-11-20'::date, '2026-11-27', '1 day') d
LEFT JOIN posts p ON p.published_on = d::date
GROUP BY d ORDER BY d;
```

```
     d      | count
------------+-------
 2026-11-20 |     3
 2026-11-21 |     3
 2026-11-22 |     2
 2026-11-23 |     5
 2026-11-24 |     6
 2026-11-25 |     4
 2026-11-26 |     4
 2026-11-27 |     5
(8 rows)
```

Enums or a check?

An enum is a type with a fixed list of values. A check constraint on a `text` column does the same job:

```
CREATE TYPE post_status AS ENUM ('draft', 'scheduled', 'published');
CREATE TEMP TABLE e1 (status post_status);
INSERT INTO e1 VALUES ('archived');
```

```
CREATE TEMP TABLE e2 (status text CHECK (status IN ('draft', 'scheduled', 'published')));
INSERT INTO e2 VALUES ('archived');


ERROR:  invalid input value for enum post_status: "archived"
LINE 1: INSERT INTO e1 VALUES ('archived');
                               ^
ERROR:  new row for relation "e2" violates check constraint "e2_status_check"
DETAIL:  Failing row contains (archived).
```

Both refuse the value. The difference shows up when the list changes. Adding a value to an enum is one quick command:

```
ALTER TYPE post_status ADD VALUE 'archived';
SELECT enum_range(NULL::post_status);


              enum_range
-------------------------------------
 {draft,scheduled,published,archived}
(1 row)
```

But you can't remove a value from an enum. You'd have to create a new type and convert the column. A check constraint you can drop and recreate any time. I use enums for lists that only grow, and checks for everything else.

Picking a type

For	Use	Avoid
Text of any length	`text`	`char(n)`
Text with a real limit	`text` with a `CHECK`	`varchar(n)` if the limit may change
A moment in time	`timestamptz`	`timestamp`
A calendar day	`date`	a timestamp at midnight
Money	`numeric(12,2)` or `bigint` cents	`float8`, `money`
Generated IDs	`bigint` identity, or `uuid` with `uuidv7()`	`integer`, `uuidv4()` as a key
A fixed list that only grows	`enum`	
A list that changes	`text` with a `CHECK`, or a lookup table	`enum`
A start and an end	a range type	two columns, if you need overlap checks

Identity columns, sequences and constraints

Every table needs a primary key, and most of the time it's a number the database generates. PostgreSQL has two ways to do that, an old one and a new one.

serial and identity

The old way is `serial`. It's not a real type. It creates an integer column, a sequence, and a default that calls `nextval()` on the sequence.

The new way, since PostgreSQL 10, is an **identity column**, which is the SQL standard. My posts table uses one:

```
id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY
```

The difference shows up when someone inserts an ID by hand. Let's try both:

```
CREATE TEMP TABLE s1 (id serial PRIMARY KEY, name text);
CREATE TEMP TABLE s2 (id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY, name text);

INSERT INTO s1 (id, name) VALUES (1, 'manual');
INSERT INTO s1 (name) VALUES ('auto');
```

```
INSERT 0 1
ERROR:  duplicate key value violates unique constraint "s1_pkey"
DETAIL:  Key (id)=(1) already exists.
```

The `serial` table accepted my manual `1`, and its sequence didn't know about it. The next automatic insert asked the sequence for a number, got `1`, and crashed into the row I'd inserted. In a real application this error shows up hours later, in a request that has nothing to do with the manual insert.

With `GENERATED ALWAYS`, the manual insert doesn't get through:

```
INSERT INTO s2 (id, name) VALUES (1, 'manual');
```

```
ERROR:  cannot insert a non-DEFAULT value into column "id"
DETAIL:  Column "id" is an identity column defined as GENERATED ALWAYS.
HINT:  Use OVERRIDING SYSTEM VALUE to override.
```

You can still force it when you mean it, with `OVERRIDING SYSTEM VALUE`. But then you're in the same situation as with `serial`: the next automatic insert fails with the same duplicate key error. Overriding is a deliberate act, at least.

	`serial`	`GENERATED ALWAYS AS IDENTITY`	`GENERATED BY DEFAULT AS IDENTITY`
SQL standard	no	yes	yes

	serial	GENERATED ALWAYS AS IDENTITY	GENERATED BY DEFAULT AS IDENTITY
Manual values	accepted silently	refused, unless OVERRIDING SYSTEM VALUE	accepted silently
Needs a grant on the sequence	yes	no	no
Sequence owned by the column	yes	yes	yes

I use `GENERATED ALWAYS` for new tables. `BY DEFAULT` is useful when you import data with existing IDs.

Fixing a sequence after an import

Importing rows with their own IDs is the classic way to break a sequence. Here I use `BY DEFAULT`, insert three rows with IDs, then let the database pick:

```
CREATE TEMP TABLE imp (id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, name text);
INSERT INTO imp (id, name) VALUES (1, 'a'), (2, 'b'), (3, 'c');
INSERT INTO imp (name) VALUES ('d');
```

```
INSERT 0 3
ERROR:  duplicate key value violates unique constraint "imp_pkey"
DETAIL:  Key (id)=(1) already exists.
```

The sequence is still at the start. The fix is to move it past the highest ID. `pg_get_serial_sequence` finds the sequence behind a column, for `serial` and identity alike, and `setval` moves it:

```
SELECT setval(pg_get_serial_sequence('imp', 'id'), (SELECT max(id) FROM imp));
INSERT INTO imp (name) VALUES ('d') RETURNING *;
```

```
 setval
--------
      3
(1 row)

 id | name
----+------
  4 | d
(1 row)
```

For an identity column there's also a shorter way to set the next value:

```
ALTER TABLE imp ALTER COLUMN id RESTART WITH 100;
```

Sequences have gaps, and that's fine

A sequence hands out numbers and never takes them back. If the transaction that got a number rolls back, the number is lost:

```
CREATE TEMP TABLE g (id int GENERATED ALWAYS AS IDENTITY, x int);
INSERT INTO g (x) VALUES (1);
BEGIN; INSERT INTO g (x) VALUES (2); ROLLBACK;
INSERT INTO g (x) VALUES (3);
SELECT * FROM g;
```

```
 id | x
----+---
  1 | 1
  3 | 3
(2 rows)
```

Same for an insert that fails a constraint. Later in this chapter a failed insert into `posts` used up ID 2194.

This is on purpose. If sequences waited to see whether each transaction commits, every insert in the database would have to wait for the one before. So don't use IDs for anything that must have no gaps, like invoice numbers. Generate those yourself, inside the transaction that creates the invoice.

Foreign keys and ON DELETE

A foreign key makes sure a value points to a row that exists. What happens when that row is deleted depends on the `ON DELETE` action. My `post_tags` uses `ON DELETE CASCADE`, so deleting a post also deletes its tag links:

```
BEGIN;
SELECT count(*) AS links_before FROM post_tags;
DELETE FROM posts WHERE slug = 'postgres-introduction';
SELECT count(*) AS links_after FROM post_tags;
ROLLBACK;
```

```
 links_before
--------------
         2239
(1 row)

DELETE 1
 links_after
-------------
        2238
(1 row)
```

The default, `NO ACTION`, refuses the delete while rows still point to it. And the foreign key refuses an insert that points nowhere:

```
CREATE TEMP TABLE parent (id int PRIMARY KEY);
CREATE TEMP TABLE child (pid int REFERENCES parent (id));
INSERT INTO parent VALUES (1);
INSERT INTO child VALUES (1);
DELETE FROM parent WHERE id = 1;
INSERT INTO child VALUES (2);
```

```
ERROR:  update or delete on table "parent" violates foreign key constraint "child_pid_fkey" on table "
    ↪ child"
DETAIL:  Key (id)=(1) is still referenced from table "child".
ERROR:  insert or update on table "child" violates foreign key constraint "child_pid_fkey"
DETAIL:  Key (pid)=(2) is not present in table "parent".
```

ON DELETE	When the parent row is deleted
NO ACTION (default)	the delete fails if child rows exist
RESTRICT	the same, checked immediately even in deferred mode
CASCADE	the child rows are deleted too
SET NULL	the child column becomes NULL
SET DEFAULT	the child column gets its default

PostgreSQL creates an index for primary keys and unique constraints, but **not** for the referencing side of a foreign key. `post_tags.post_id` is covered by the primary key `(post_id, tag_id)`, but in the indexes chapter we'll see what happens on a table where it isn't.

Constraints checked at the end of a statement

Here's something I didn't expect. Let's swap two values in a unique column, in one `UPDATE`:

```
CREATE TEMP TABLE pos2 (id int PRIMARY KEY, n int UNIQUE);
INSERT INTO pos2 VALUES (1, 1), (2, 2);
UPDATE pos2 SET n = 3 - n;
```

```
ERROR:  duplicate key value violates unique constraint "pos2_n_key"
DETAIL:  Key (n)=(2) already exists.
```

After the statement, the values are unique again (2 and 1). But PostgreSQL checks a normal unique constraint row by row, and after the first row changes to 2, there are two 2s for a moment.

Make the constraint `DEFERRABLE`, and the same statement works:

```
CREATE TEMP TABLE pos (id int PRIMARY KEY, n int UNIQUE DEFERRABLE INITIALLY IMMEDIATE);
INSERT INTO pos VALUES (1, 1), (2, 2);
UPDATE pos SET n = 3 - n;
```

```
UPDATE 2
```

A deferrable constraint, even in `INITIALLY IMMEDIATE` mode, is checked at the end of the statement. And with `SET CONSTRAINTS ALL DEFERRED` inside a transaction, the check moves to the `COMMIT`, so you can break it across several statements as long as everything is fine at the end.

Exclusion constraints

A unique constraint says "no two rows have equal values". An exclusion constraint generalizes it: "no two rows have values that conflict", with the operator of your choice.

The classic example is bookings. Two bookings of the same room must not overlap in time:

```
CREATE EXTENSION IF NOT EXISTS btree_gist;

CREATE TEMP TABLE bookings (
  room text,
  during tstzrange,
  EXCLUDE USING gist (room WITH =, during WITH &&)
);

INSERT INTO bookings VALUES ('a', '[2026-12-01 10:00, 2026-12-01 11:00)');
INSERT INTO bookings VALUES ('a', '[2026-12-01 10:30, 2026-12-01 12:00)');
INSERT INTO bookings VALUES ('b', '[2026-12-01 10:30, 2026-12-01 12:00)');
```

```
INSERT 0 1
ERROR:  conflicting key value violates exclusion constraint "bookings_room_during_excl"
DETAIL:  Key (room, during)=(a, ["2026-12-01 10:30:00+01","2026-12-01 12:00:00+01")) conflicts with
    ↪ existing key (room, during)=(a, ["2026-12-01 10:00:00+01","2026-12-01 11:00:00+01")).
INSERT 0 1
```

Same room and overlapping times (`&&`): refused. Another room at the same time: fine. Checking this in application code is surprisingly hard to get right when two requests arrive at once. The database gets it right every time. (`btree_gist` is a standard extension that lets the `=` on `text` live in the same index as the range.)

Check constraints on existing data

A `CHECK` constraint added to a table with data is verified against every row:

```
ALTER TABLE posts ADD CONSTRAINT words_positive CHECK (words >= 0);
INSERT INTO posts (slug, title, published_on, words, body, frontmatter)
VALUES ('x', 'x', '2026-01-01', -1, '', '{}');
```

```
ALTER TABLE
ERROR:  new row for relation "posts" violates check constraint "words_positive"
DETAIL:  Failing row contains (2194, x, x, 2026-01-01, -1, , {}).
```

There's the ID 2194 that was used up.

On a big table, verifying every row takes time, and the table is locked while it happens. `NOT VALID` adds the constraint for new rows only, and `VALIDATE CONSTRAINT` checks the old rows later, with a lighter lock that lets reads and writes go on:

```
ALTER TABLE posts ADD CONSTRAINT slug_lowercase CHECK (slug = lower(slug)) NOT VALID;
ALTER TABLE posts VALIDATE CONSTRAINT slug_lowercase;
```

PostgreSQL 18 can do the same with `NOT NULL`, which until now always needed a full check while locked:

```
ALTER TABLE tags ADD CONSTRAINT tags_name_nn NOT NULL name NOT VALID;
ALTER TABLE tags VALIDATE CONSTRAINT tags_name_nn;
```

We'll measure the difference in the migrations chapter.

RETURNING and upserts

These are two features I use in almost every application, and both save a round trip to the database.

RETURNING

`INSERT`, `UPDATE` and `DELETE` can return the rows they changed. The most common use is getting the generated ID of a new row:

```
INSERT INTO tags (name) VALUES ('sqlite-vec') RETURNING id, name;
```

```
 id |    name
----+------------
 51 | sqlite-vec
(1 row)
```

Without `RETURNING`, you'd need a second query to find out the ID, and you'd have to be careful that it finds *your* row.

`DELETE ... RETURNING` tells you exactly what you removed:

```
DELETE FROM tags WHERE name IN ('sqlite-vec', 'postgresql') RETURNING *;
```

```
 id |    name
----+------------
 50 | postgresql
 51 | sqlite-vec
(2 rows)
```

OLD and NEW in PostgreSQL 18

Until PostgreSQL 17, an `UPDATE ... RETURNING` could only return the new values. PostgreSQL 18 adds `old` and `new`, so one statement tells you what changed:

```
UPDATE posts SET words = words + 10
WHERE slug = 'postgres-introduction'
RETURNING old.words AS before, new.words AS after;
```

```
 before | after
--------+-------
    785 |   795
(1 row)
```

That's an audit log entry, or a "you changed X from A to B" message, without reading the row first.

Upserts with ON CONFLICT

An **upsert** inserts a row, or updates it if it's already there. Page view counters are the textbook case. I made a small table for them:

```
CREATE TABLE page_views (
  slug text PRIMARY KEY REFERENCES posts (slug),
  views bigint NOT NULL DEFAULT 0,
  updated_at timestamptz NOT NULL DEFAULT now()
);
```

My first try was with `'docker'`, the slug of my Docker guide, and it failed:

```
ERROR:  insert or update on table "page_views" violates foreign key constraint "page_views_slug_fkey"
DETAIL:  Key (slug)=(docker) is not present in table "posts".
```

The Docker guide is scheduled for December 2026, so my loader skipped it, and the foreign key caught it. The constraint did its job, and I switched to the Claude Code guide.

A plain insert works the first time and fails the second:

```
INSERT 0 1
ERROR:  duplicate key value violates unique constraint "page_views_pkey"
DETAIL:  Key (slug)=(claude-code-guide) already exists.
```

`ON CONFLICT ... DO UPDATE` turns the second insert into an update:

```
INSERT INTO page_views (slug, views) VALUES ('claude-code-guide', 1)
ON CONFLICT (slug) DO UPDATE
  SET views = page_views.views + excluded.views, updated_at = now()
RETURNING slug, views, (xmax = 0) AS inserted;
```

```
       slug        | views | inserted
-------------------+-------+----------
 claude-code-guide |     2 | f
(1 row)
```

`excluded` is the row you tried to insert. `page_views.views` is the one already in the table. The `(xmax = 0)` trick tells you whether the row was inserted or updated (it relies on how MVCC works, which we'll see later). For a new slug it says `t`:

```
   slug    | views | inserted
-----------+-------+----------
 tailscale |     1 | t
(1 row)
```

With PostgreSQL 18, `old` and `new` work here too, and `old` is `NULL` when the row was inserted:

```
INSERT INTO page_views (slug, views) VALUES ('claude-code-guide', 5)
ON CONFLICT (slug) DO UPDATE SET views = page_views.views + excluded.views
RETURNING old.views AS before, new.views AS after;
```

```
 before | after
--------+-------
      2 |     7
(1 row)
```

The big advantage over "select, then insert or update" in application code is that the upsert is atomic. Two requests at the same time can't both decide the row is missing and both insert it.

`DO NOTHING` skips the conflicting row. Then `RETURNING` returns nothing for it, which is how you can tell:

```
INSERT INTO page_views (slug) VALUES ('claude-code-guide') ON CONFLICT DO NOTHING RETURNING *;
```

```
 slug | views | updated_at
------+-------+------------
(0 rows)

INSERT 0 0
```

One limit to know: a single statement can't hit the same row twice.

```
INSERT INTO page_views (slug, views) VALUES ('htmx', 1), ('htmx', 1)
ON CONFLICT (slug) DO UPDATE SET views = page_views.views + excluded.views;
```

```
ERROR:  ON CONFLICT DO UPDATE command cannot affect row a second time
HINT:  Ensure that no rows proposed for insertion within the same command have duplicate constrained
    ↪ values.
```

When you upsert a batch, group the duplicates first (`SELECT slug, sum(views) ... GROUP BY slug`).

MERGE

`MERGE` is the SQL standard way to sync one table into another, with a different action for rows that match and rows that don't. Here I merge one day of views into the counters. The source has a slug that isn't a post, so I filter it with a join first:

```
CREATE TEMP TABLE daily (slug text, views int);
INSERT INTO daily VALUES ('claude-code-guide', 10), ('htmx', 3), ('not-a-post', 7);

MERGE INTO page_views pv
USING (SELECT d.* FROM daily d JOIN posts p ON p.slug = d.slug) d
ON pv.slug = d.slug
WHEN MATCHED THEN UPDATE SET views = pv.views + d.views
WHEN NOT MATCHED THEN INSERT (slug, views) VALUES (d.slug, d.views)
RETURNING merge_action(), pv.slug, pv.views;


 merge_action |       slug        | views
--------------+-------------------+-------
 UPDATE       | claude-code-guide |    17
 INSERT       | htmx              |     3
(2 rows)

MERGE 2
```

`merge_action()` and `RETURNING` on `MERGE` came in PostgreSQL 17. `MERGE` can also delete:

```
MERGE INTO page_views pv
USING (VALUES ('tailscale')) AS gone(slug) ON pv.slug = gone.slug
WHEN MATCHED THEN DELETE
RETURNING merge_action(), old.slug, old.views;


 merge_action |   slug    | views
--------------+-----------+-------
 DELETE       | tailscale |     1
(1 row)
```

For a single row I stick with `ON CONFLICT`, which is shorter and handles concurrent inserts better. `MERGE` is for syncing a batch, especially when some rows must be deleted.

JSONB

Every post on my blog starts with frontmatter: a few lines of YAML with the title, the date, the tags, and other fields that vary from post to post. I loaded it into the `frontmatter` column as `jsonb`, which is JSON stored in a binary form PostgreSQL can search and index.

What's in there

`jsonb_object_keys` lists the keys of an object, one per row. Grouping them shows which fields my posts use:

```
SELECT key, count(*)
FROM posts, jsonb_object_keys(frontmatter) AS key
GROUP BY key ORDER BY 2 DESC;
```

```
       key       | count
-----------------+-------
 tag             |  2193
 date            |  2193
 description     |  2193
 title           |  2193
 updated         |  1824
 ogImage         |   392
 tags_weight     |   126
 path            |    81
 widget          |    28
 noindex         |    24
 small           |    11
 obsolete        |     7
 bookdescription |     5
 math            |     4
 app             |     3
 so              |     3
 invertimages    |     2
 smaller         |     2
 draftKey        |     2
 sponsor         |     1
 featured        |     1
 sponsorUrl      |     1
 ogDescription   |     1
 seo             |     1
(24 rows)
```

Four fields in every post, and twenty more that show up now and then. That's the case where JSON makes sense: twenty sparse columns would be mostly `NULL`.

-> and -»

`->` gets a value as JSON, and `->>` gets it as text:

```
SELECT frontmatter -> 'tag' AS json, frontmatter ->> 'tag' AS text,
       jsonb_typeof(frontmatter -> 'tag')
FROM posts WHERE slug = 'the-aha-stack';
```

```
            json             |            text             | jsonb_typeof
-----------------------------+-----------------------------+--------------
 ["astro", "htmx", "alpine"] | ["astro", "htmx", "alpine"] | array
(1 row)
```

That `array` was a surprise. In my frontmatter, `tag` is sometimes one string and sometimes a list:

```
SELECT jsonb_typeof(frontmatter -> 'tag') AS type, count(*) FROM posts GROUP BY 1;
```

```
  type  | count
--------+-------
 string |  2086
 array  |   107
(2 rows)
```

Real data is messy like this, and it matters for every query on that field.

`->>` returns text, so cast it when you need another type. `?` checks whether a key exists:

```
SELECT slug, (frontmatter ->> 'updated')::date - published_on AS days_later
FROM posts
WHERE frontmatter ? 'updated'
ORDER BY 2 DESC LIMIT 5;
```

```
                  slug                    | days_later
------------------------------------------+------------
 things-to-avoid-in-javascript-bad-parts |       5179
 introduction-to-yeoman                  |       4887
 bower-package-manager                   |       4863
 frontend-testing                        |       4847
 deferreds-and-promises-in-javascript    |       4814
(5 rows)
```

My most-updated old post was revised 5,179 days after I published it.

Containment with @>

`@>` asks ”does the left JSON contain the right one?”. It's the operator to use for filtering, because it can use an index. How many posts are tagged `ai`?

```
SELECT count(*) FROM posts WHERE frontmatter @> '{"tag": "ai"}';
SELECT count(*) FROM posts WHERE frontmatter @> '{"tag": ["ai"]}';
```

```
 count
-------
   204
(1 row)

 count
-------
     7
(1 row)
```

204 posts have `tag: ai` as a string. 7 more have `ai` inside a list. Containment matches the shape exactly, so to get both I'd need two conditions with `OR`.

A JSON path query is more forgiving. In its default "lax" mode, `[*]` treats a single value like a list of one:

```
SELECT count(*) FROM posts WHERE frontmatter @? '$.tag[*] ? (@ == "ai")';
```

```
 count
-------
   211
(1 row)
```

204 + 7 = 211, in one condition. Here's the lax mode at work on the two shapes:

```
SELECT jsonb_path_query_array('{"tag":"ai"}', '$.tag[*]'),
       jsonb_path_query_array('{"tag":["ai","node"]}', '$.tag[*]');
```

```
 jsonb_path_query_array | jsonb_path_query_array
------------------------+------------------------
 ["ai"]                 | ["ai", "node"]
(1 row)
```

Indexing JSONB

Without an index, a containment query reads the whole table. `EXPLAIN ANALYZE`, which we'll study properly in a later chapter, shows how a query ran:

```
EXPLAIN ANALYZE SELECT count(*) FROM posts WHERE frontmatter @> '{"tag": "docker"}';
```

```
 Aggregate  (cost=258.42..258.43 rows=1 width=8) (actual time=0.665..0.665 rows=1.00 loops=1)
   Buffers: shared hit=231
   ->  Seq Scan on posts  (cost=0.00..258.41 rows=1 width=0) (actual time=0.224..0.660 rows=17.00 loops
     ↪ =1)
         Filter: (frontmatter @> '{"tag": "docker"}'::jsonb)
         Rows Removed by Filter: 2176
         Buffers: shared hit=231
 Planning Time: 0.273 ms
 Execution Time: 0.688 ms
```

`Seq Scan` means it read every row and threw away 2,176 of them. A GIN index fixes that. GIN indexes the inside of a value, every key and every element, instead of the value as a whole:

```
CREATE INDEX posts_frontmatter_gin ON posts USING gin (frontmatter jsonb_path_ops);
ANALYZE posts;
```

`jsonb_path_ops` is a variant that only supports `@>` and the path operators, and makes a smaller, faster index. The same query now:

```
 Aggregate  (cost=16.83..16.84 rows=1 width=8) (actual time=0.029..0.029 rows=1.00 loops=1)
   Buffers: shared hit=19
   ->  Bitmap Heap Scan on posts  (cost=12.82..16.83 rows=1 width=0) (actual time=0.015..0.025 rows
    ↪ =17.00 loops=1)
         Recheck Cond: (frontmatter @> '{"tag": "docker"}'::jsonb)
         Rows Removed by Index Recheck: 5
         Heap Blocks: exact=16
         Buffers: shared hit=19
         ->  Bitmap Index Scan on posts_frontmatter_gin  (cost=0.00..12.82 rows=1 width=0) (actual time
    ↪ =0.008..0.008 rows=22.00 loops=1)
               Index Cond: (frontmatter @> '{"tag": "docker"}'::jsonb)
               Index Searches: 1
               Buffers: shared hit=3
 Planning Time: 0.207 ms
 Execution Time: 0.058 ms
```

From 0.688 ms to 0.058 ms, and from 231 pages read to 19. On a table of 2,193 rows both are fast. On a table of millions, it's the difference between a working page and a timeout. The index takes 424 kB.

Changing JSONB values

There are operators to build and modify JSON:

```
SELECT jsonb_set('{"title":"x"}', '{draft}', 'true'),
       '{"a":1}'::jsonb || '{"b":2}',
       '{"a":1,"b":2}'::jsonb - 'a';
```

```
           jsonb_set            |     ?column?     | ?column?
--------------------------------+------------------+----------
 {"draft": true, "title": "x"} | {"a": 1, "b": 2} | {"b": 2}
(1 row)
```

`jsonb_set` sets a path, `||` merges two objects, and `-` removes a key.

json or jsonb

PostgreSQL has both. `json` stores the text as you gave it. `jsonb` parses it:

```
SELECT '{"b":1, "a":2, "a":3}'::json AS json, '{"b":1, "a":2, "a":3}'::jsonb AS jsonb;
```

```
         json          |      jsonb
-----------------------+------------------
 {"b":1, "a":2, "a":3} | {"a": 3, "b": 1}
(1 row)
```

	`json`	`jsonb`
Storage	the original text	a parsed binary form
Keys order and whitespace	kept	not kept
Duplicate keys	kept	the last one wins

	`json`	`jsonb`
Writing	a little faster	a little slower (parsing)
Reading and searching	slower, parsed every time	fast
GIN indexes, `@>`, `?`	no	yes

I always use `jsonb`. The only reason for `json` is when you must store the exact text you received, like a webhook payload you may need to verify.

Turning JSON back into columns

`jsonb_to_record` reads fields as typed columns:

```
SELECT r.*
FROM posts, jsonb_to_record(frontmatter) AS r(title text, date timestamptz, description text)
WHERE slug = 'claude-code-guide';


              title              |          date          |
    ↪                                                 description
-----------------------------------+------------------------+------------------------------------------
 The complete guide to Claude Code | 2026-11-23 09:25:00+01 | The complete handbook to Claude Code v2
    ↪ .1: agent loop, prompt caching, models, CLAUDE.md and rules, permissions, sandboxing, parallel
    ↪ agents, hooks, skills, MCP, plugins, CI and the Agent SDK.
(1 row)
```

PostgreSQL 17 added the SQL standard functions too, like `JSON_TABLE`:

```
SELECT jt.*
FROM posts,
  JSON_TABLE(frontmatter, '$' COLUMNS (
    title text PATH '$.title',
    updated date PATH '$.updated'
  )) AS jt
WHERE slug = 'claude-code-guide';


               title               | updated
-----------------------------------+---------
 The complete guide to Claude Code |
(1 row)
```

`updated` is empty because that post has never been updated.

When to use a real column

JSONB is flexible, and that's the risk. Here's my rule:

- A field every row has, that you filter, sort or join on, becomes a real column. `title` and `date` are columns in my table, even if they're also in the JSON.
- A field that needs a constraint (not null, unique, a foreign key) becomes a column. JSON can't have foreign keys.
- Fields that are optional, rare or different from row to row stay in JSONB.

Full-text search

Searching text with `ILIKE` works, but it's not search. Let's see why, then build the real thing on the 2 million words of my posts.

What's wrong with ILIKE

`ILIKE '%index%'` matches the letters, not the word:

```
SELECT count(*) FILTER (WHERE body ILIKE '%index%') AS ilike_index,
       count(*) FILTER (WHERE body ILIKE '%indexes%') AS ilike_indexes
FROM posts;
```

```
 ilike_index | ilike_indexes
-------------+---------------
         429 |            79
(1 row)
```

The first one also matches `index.html`, `indexOf` and `reindex`. The second misses posts that only say ”index” or ”indexing”. It can't rank results, so the post about indexes and a post that mentions `index.js` once come out the same. And a `%` at the start means no normal index can help.

tsvector and tsquery

Full-text search works on **lexemes**: words reduced to a normalized root. `to_tsvector` turns text into a sorted list of lexemes and their positions:

```
SELECT to_tsvector('english', 'Running containers with Docker Compose, and indexing the indexes');
```

```
                      to_tsvector
-------------------------------------------------------
 'compos':5 'contain':2 'docker':4 'index':7,9 'run':1
(1 row)
```

”Running” became `run`, ”indexing” and ”indexes” both became `index`, and the common words ”with”, ”and” and ”the” (called **stop words**) are gone. The `english` configuration decides all of this. With the `simple` configuration, nothing is reduced:

```
                        to_tsvector
-----------------------------------------------------------
 'compose':5 'containers':2 'docker':4 'running':1 'with':3
(1 row)
```

A `tsquery` is the search side, normalized the same way. `@@` checks if one matches the other:

```
SELECT to_tsvector('english', 'I indexed the table') @@ to_tsquery('english', 'indexes');
```

```
 ?column?
----------
 t
(1 row)
```

"indexed" and "indexes" match, because both are `index`.

`to_tsquery` needs its own syntax with `&` and `|`. For search boxes, `websearch_to_tsquery` reads what people type, with quotes for phrases and `-` to exclude:

```
SELECT websearch_to_tsquery('english', 'docker compose -swarm "multi-stage build"');
```

```
                              websearch_to_tsquery
-------------------------------------------------------------------------------
 'docker' & 'compos' & !'swarm' & 'multi-stag' <-> 'multi' <-> 'stage' <-> 'build'
(1 row)
```

`<->` means "followed by", for the phrase.

Searching without an index

The simplest version computes the `tsvector` of every post at query time:

```
\timing on
SELECT count(*) FROM posts
WHERE to_tsvector('english', body) @@ websearch_to_tsquery('english', 'postgres index');
```

```
NOTICE:  word is too long to be indexed
DETAIL:  Words longer than 2047 characters are ignored.
...
 count
-------
    23
(1 row)

Time: 1099.462 ms (00:01.099)
```

Over a second. And a screen full of notices: some word in my posts is longer than 2,047 characters. Which one?

```
SELECT slug, (SELECT max(length(w)) FROM regexp_split_to_table(body, '\s+') w) AS longest
FROM posts ORDER BY 2 DESC LIMIT 2;
```

```
                   slug                    | longest
-------------------------------------------+---------
 data-urls                                 |   10670
 how-to-validate-email-address-javascript  |     400
(2 rows)
```

My post on data URLs has a base64-encoded image in it, 10,670 characters with no spaces. PostgreSQL skips it and tells you. It's harmless, but now you know what the notice means when you see it.

A stored generated column and a GIN index

Computing the `tsvector` once per row, and indexing it, is the real setup. A **generated column** keeps it in sync with the text automatically.

Here's PostgreSQL 18 surprising me again:

```
ALTER TABLE posts ADD COLUMN search_v tsvector
  GENERATED ALWAYS AS (to_tsvector('english', title || ' ' || body));
CREATE INDEX posts_search_v_idx ON posts USING gin (search_v);


ALTER TABLE
ERROR:  indexes on virtual generated columns are not supported
```

PostgreSQL 18 added **virtual** generated columns, computed when you read them instead of stored, and made them the default. A virtual column can't be indexed. You need to write `STORED`.

The other trap: `to_tsvector(body)` with one argument depends on the `default_text_search_config` setting, which can change, so PostgreSQL refuses it in a generated column:

```
ERROR:  generation expression is not immutable
```

Always pass the configuration. Here's the version that works, with the title weighted higher than the body (`A` is the highest weight, `D` the lowest):

```
\timing on
ALTER TABLE posts ADD COLUMN search tsvector
  GENERATED ALWAYS AS (
    setweight(to_tsvector('english', title), 'A') ||
    setweight(to_tsvector('english', body), 'B')
  ) STORED;
CREATE INDEX posts_search_idx ON posts USING gin (search);


ALTER TABLE
Time: 1290.976 ms (00:01.291)
CREATE INDEX
Time: 216.924 ms
```

1.3 seconds to compute the vectors of 2,193 posts, and 0.2 seconds for the index. The posts table, with its indexes, grew from 10 MB to 29 MB, and the new index alone takes 6.3 MB.

The same search again:

```
SELECT count(*) FROM posts WHERE search @@ websearch_to_tsquery('english', 'postgres index');


 count
-------
    23
(1 row)

Time: 1.455 ms
```

From 1,099 ms to 1.5 ms. The plan shows the GIN index at work:

```
 Aggregate  (cost=34.06..34.07 rows=1 width=8) (actual time=0.051..0.051 rows=1.00 loops=1)
   Buffers: shared hit=27
   ->  Bitmap Heap Scan on posts  (cost=30.05..34.06 rows=1 width=0) (actual time=0.025..0.047 rows
    ↪ =23.00 loops=1)
         Recheck Cond: (search @@ '''postgr'' & ''index'''::tsquery)
         Heap Blocks: exact=20
         Buffers: shared hit=27
         ->  Bitmap Index Scan on posts_search_idx  (cost=0.00..30.05 rows=1 width=0) (actual time
    ↪ =0.019..0.019 rows=23.00 loops=1)
               Index Cond: (search @@ '''postgr'' & ''index'''::tsquery)
               Index Searches: 1
               Buffers: shared hit=7
 Planning Time: 1.124 ms
 Execution Time: 0.072 ms
```

Postgres is not PostgreSQL

Look at the query in that plan: `'postgr' & 'index'`. Only 23 posts matched, which seemed low for a blog with a dozen PostgreSQL tutorials. So I checked how the stemmer reads the name:

```
SELECT to_tsvector('english', 'Postgres PostgreSQL postgres-introduction');


                            to_tsvector
------------------------------------------------------------------
 'introduct':5 'postgr':1,4 'postgres-introduct':3 'postgresql':2
(1 row)
```

"Postgres" becomes `postgr` and "PostgreSQL" stays `postgresql`. To the search engine they're two unrelated words. Counting the three versions of the query:

```
SELECT
  count(*) FILTER (WHERE search @@ websearch_to_tsquery('english', 'postgres index')) AS a,
  count(*) FILTER (WHERE search @@ websearch_to_tsquery('english', 'postgresql index')) AS b,
  count(*) FILTER (WHERE search @@ websearch_to_tsquery('english', '(postgres or postgresql) index'))
    ↪ AS c
FROM posts;


 a  | b  | c
----+----+----
 23 | 24 | 88
(1 row)
```

23 and 24, but 88 with both names. The stemmer works on grammar, not meaning. For names and synonyms you either expand the query yourself, like I did here, or add a synonym dictionary to the text search configuration.

Ranking

`ts_rank` scores each match: more occurrences and higher weights mean a higher score. Here are the best matches for "postgresql index":

```
SELECT slug, round(ts_rank(search, query)::numeric, 3) AS rank
FROM posts, websearch_to_tsquery('english', 'postgresql index') AS query
WHERE search @@ query
ORDER BY rank DESC
LIMIT 6;
```

```
             slug              | rank
-------------------------------+-------
 llm-hallucinations-sycophancy | 0.786
 kysely                        | 0.578
 postgres-introduction         | 0.503
 cloudflare-hyperdrive         | 0.428
 doctl                         | 0.350
 do-you-need-a-database        | 0.119
(6 rows)
```

A post about LLM hallucinations comes first, which is not what a reader searching "postgresql index" wants. Ranking is a heuristic. It's good enough for a site search, and worth checking with your own queries. Weights, `ts_rank_cd` (which rewards words close together), and a normalization option that penalizes long documents are the knobs to try.

Snippets

`ts_headline` shows a piece of the text around the matches, with markers you choose. It's what a search results page needs:

```
SELECT slug, ts_headline('english', body, query,
  'MaxWords=18, MinWords=8, StartSel=[, StopSel=]') AS snippet
FROM posts, websearch_to_tsquery('english', 'vacuum') AS query
WHERE search @@ query
ORDER BY ts_rank(search, query) DESC
LIMIT 3;
```

```
             slug              |                      snippet
-------------------------------+-----------------------------------------------
 node-sqlite                   | [VACUUM] INTO`:                               +
                               |                                               +
                               | ```sql                                        +
                               | [VACUUM] INTO '/backups/notes-2026-11-28.db'; +
                               | ```                                           +
```

```
                              |                                                 +
                              | Or run the `sqlite3
 linux-server-commands        | [vacuum]-time` and the other cheap cleanups     +
                              |                                                 +
                              | Before
 create-empty-sqllite-database | [VACUUM];"                                      +
                              | ```                                             +
                              |                                                 +
                              | The difference is small but visible. Run `file
(3 rows)
```

`ts_headline` works on the original text, so it's slow on long documents. Run it only on the rows you display, after the `LIMIT`, like here. And clean the Markdown out of the text if your snippets go in a web page.

More query types

Phrases need the words next to each other. Without quotes, both words anywhere in the post are enough:

```
SELECT count(*) FROM posts WHERE search @@ websearch_to_tsquery('english', '"connection pool"');
SELECT count(*) FROM posts WHERE search @@ websearch_to_tsquery('english', 'connection pool');
```

```
 count
-------
    20
(1 row)

 count
-------
    53
(1 row)
```

`:*` matches a prefix, which is how you'd do search-as-you-type:

```
SELECT count(*) FROM posts WHERE search @@ to_tsquery('english', 'postg:*');
```

```
 count
-------
   121
(1 row)
```

A query made only of stop words matches nothing, and PostgreSQL tells you:

```
NOTICE:  text-search query contains only stop words or doesn't contain lexemes, ignored
```

`ts_stat` shows the most common lexemes across all documents. On my posts it's a portrait of how I write:

```
SELECT word, ndoc FROM ts_stat('SELECT search FROM posts') ORDER BY ndoc DESC LIMIT 10;
```

```
 word  | ndoc
-------+------
 use   | 2086
 one   | 1873
 work  | 1705
 need  | 1647
 like  | 1556
 also  | 1518
 first | 1481
 get   | 1459
 run   | 1387
 want  | 1385
(10 rows)
```

”use” appears in 2,086 of 2,193 posts.

When PostgreSQL search is enough

For a blog, documentation, or the search box of most applications, this chapter is all you need: one column, one index, one query, no extra service to run and keep in sync. I'd move to a dedicated search engine when I need typo tolerance, faceted filters on millions of documents, or relevance tuning beyond weights.

EXPLAIN ANALYZE

When a query is slow, guessing why is a waste of time. PostgreSQL tells you exactly what it did, if you ask with `EXPLAIN`.

EXPLAIN and EXPLAIN ANALYZE

`EXPLAIN` shows the **plan**: the steps PostgreSQL intends to take, with its estimates. It doesn't run the query:

```
EXPLAIN SELECT * FROM posts WHERE id = 732;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using posts_pkey on posts  (cost=0.28..8.30 rows=1 width=799)
   Index Cond: (id = 732)
(2 rows)
```

`EXPLAIN ANALYZE` runs the query for real and adds what happened: actual times and actual row counts. Be careful with it on `INSERT`, `UPDATE` and `DELETE`, because they really change data. Wrap them in `BEGIN` and `ROLLBACK`.

```
EXPLAIN ANALYZE SELECT id, title FROM posts WHERE slug = 'postgres-introduction';
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------
```

```
 Index Scan using posts_slug_key on posts  (cost=0.28..8.30 rows=1 width=45) (actual time=0.544..0.544
    ↪ rows=1.00 loops=1)
   Index Cond: (slug = 'postgres-introduction'::text)
   Index Searches: 1
   Buffers: shared hit=2 read=1
 Planning:
   Buffers: shared hit=155
 Planning Time: 0.273 ms
 Execution Time: 0.560 ms
(8 rows)
```

Reading one line

Each line with an arrow (or the first line) is a **node**, one step of the plan. Here's how to read one:

- `Index Scan using posts_slug_key on posts`: the operation, and on what
- `cost=0.28..8.30`: the planner's estimate, in arbitrary units. The first number is the cost before the first row comes out, the second the cost to produce all of them
- `rows=1`: how many rows the planner expected
- `width=45`: the average row size in bytes
- `actual time=0.544..0.544`: milliseconds to the first row and to the last
- `rows=1.00`: how many rows actually came out (PostgreSQL 18 shows decimals, averaged over loops)
- `loops=1`: how many times this node ran

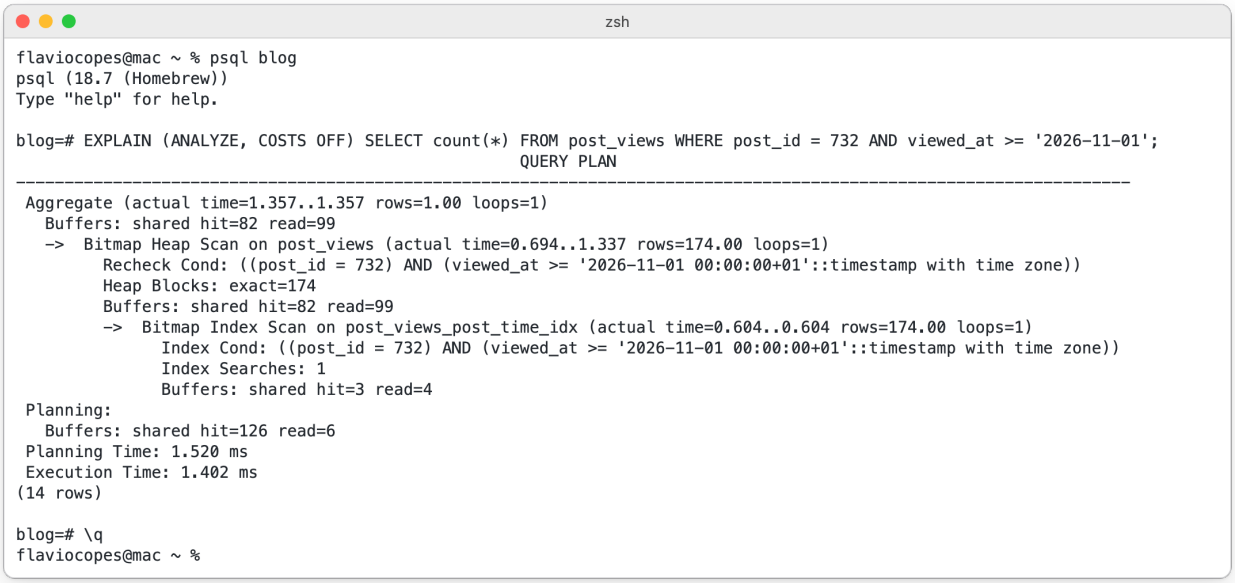
Below the node, its details. `Index Cond` is what the index was searched for. `Index Searches` is new in PostgreSQL 18, and we'll see why it matters in the indexes chapter.

Buffers

`Buffers: shared hit=2 read=1` says how many 8 kB pages the query touched: 2 were already in PostgreSQL's memory (`hit`) and 1 came from the operating system (`read`), maybe from disk.

In PostgreSQL 18 `EXPLAIN ANALYZE` includes buffers by default. Before, you had to write `EXPLAIN (ANALYZE, BUFFERS)`. Pages are often a better measure than time: timings change with what's cached, but a query that reads 37,000 pages is doing a lot of work no matter how fast your disk is.

Here's how a plan looks in a terminal, for a query on the 5-million-row table we'll build in the next chapter. With `COSTS OFF` the estimates are gone and only what happened is left:



Plans read inside out

Nodes are a tree. The most indented nodes run first and feed their rows to their parent. Here's a join across three tables: the five most recent posts tagged `ai`:

```
EXPLAIN ANALYZE
SELECT p.slug, p.published_on
FROM posts p
JOIN post_tags pt ON pt.post_id = p.id
JOIN tags t ON t.id = pt.tag_id
WHERE t.name = 'ai'
ORDER BY p.published_on DESC
LIMIT 5;
```

```
Limit  (cost=79.55..79.56 rows=5 width=26) (actual time=0.323..0.323 rows=5.00 loops=1)
  ->  Sort  (cost=79.55..79.66 rows=46 width=26) (actual time=0.322..0.322 rows=5.00 loops=1)
        Sort Key: p.published_on DESC
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Nested Loop  (cost=1.91..78.78 rows=46 width=26) (actual time=0.132..0.300 rows=211.00
    ↪ loops=1)
              ->  Hash Join  (cost=1.62..43.40 rows=46 width=8) (actual time=0.123..0.166 rows=211.00
    ↪ loops=1)
                    Hash Cond: (pt.tag_id = t.id)
                    ->  Seq Scan on post_tags pt  (cost=0.00..35.39 rows=2239 width=16) (actual time
    ↪ =0.005..0.084 rows=2239.00 loops=1)
                    ->  Hash  (cost=1.61..1.61 rows=1 width=8) (actual time=0.007..0.007 rows=1.00
    ↪ loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Seq Scan on tags t  (cost=0.00..1.61 rows=1 width=8) (actual time
    ↪ =0.002..0.004 rows=1.00 loops=1)
                                Filter: (name = 'ai'::text)
                                Rows Removed by Filter: 48
              ->  Index Scan using posts_pkey on posts p  (cost=0.28..0.77 rows=1 width=34) (actual
    ↪ time=0.001..0.001 rows=1.00 loops=211)
                    Index Cond: (id = pt.post_id)
                    Index Searches: 211
Planning Time: 1.045 ms
Execution Time: 0.341 ms
```

(I removed the `Buffers` lines to make it fit.) From the inside out:

1. Scan `tags` and keep the one row named `ai`. A sequential scan, because the table has 49 rows, smaller than one page.
2. Put that row in a hash table.
3. Scan all 2,239 rows of `post_tags` and keep the ones whose `tag_id` is in the hash: 211.
4. For each of the 211, look up the post by primary key. `loops=211` says this lookup ran 211 times, each taking 0.001 ms.
5. Sort the 211 posts by date, keeping only the top 5 in memory (`top-N heapsort`).
6. Return 5 rows.

Notice the estimate: the planner expected 46 posts tagged `ai`, and there were 211. It guessed `ai` was an average tag (2,239 links divided by 49 tags is 46). Here it doesn't matter, but wrong estimates are the root of most bad plans.

The scan types

Node	What it does	When the planner picks it
`Seq Scan`	reads every row of the table	small tables, or when many rows match
`Index Scan`	walks the index, fetches each matching row from the table	few matching rows, or rows needed in index order
`Index Only Scan`	answers from the index alone	all needed columns are in the index
`Bitmap Index Scan` + `Bitmap Heap Scan`	collects matching row locations from the index, then reads the table pages in order	a medium number of rows, or several indexes combined
`Parallel Seq Scan`	several processes split a sequential scan	big tables

And for joins:

Node	How it joins
`Nested Loop`	for each row on one side, look up matches on the other (good with an index and few rows)
`Hash Join`	build a hash table from the smaller side, probe it with the bigger one
`Merge Join`	walk two inputs sorted on the join key

A Seq Scan is not always bad

People learn ”Seq Scan bad, Index Scan good”, and it's not true. Here's a count of this year's posts:

```
EXPLAIN ANALYZE SELECT count(*) FROM posts WHERE published_on >= '2026-01-01';
```

```
 Aggregate  (cost=284.68..284.69 rows=1 width=8) (actual time=0.371..0.372 rows=1.00 loops=1)
   Buffers: shared hit=256
   ->  Seq Scan on posts  (cost=0.00..283.41 rows=507 width=0) (actual time=0.054..0.357 rows=509.00
     ↪ loops=1)
         Filter: (published_on >= '2026-01-01'::date)
         Rows Removed by Filter: 1684
         Buffers: shared hit=256
 Execution Time: 0.392 ms
```

There's no index on `published_on` yet, so a sequential scan is the only choice, and it takes 0.4 ms. When I add an index, PostgreSQL uses it:

```
CREATE INDEX posts_published_on_idx ON posts (published_on);
EXPLAIN (ANALYZE, COSTS OFF) SELECT count(*) FROM posts WHERE published_on >= '2026-01-01';
```

```
 Aggregate (actual time=0.053..0.054 rows=1.00 loops=1)
   ->  Index Only Scan using posts_published_on_idx on posts (actual time=0.009..0.041 rows=509.00
     ↪ loops=1)
         Index Cond: (published_on >= '2026-01-01'::date)
         Heap Fetches: 509
         Index Searches: 1
 Execution Time: 0.059 ms
```

`COSTS OFF` hides the estimates, which makes plans easier to read when you only care about the shape.

When a query wants a big part of a table, reading the whole table in order beats jumping around through an index. The planner makes that call with the statistics it has, and it's usually right.

Index order for free

An index on `published_on` is sorted, so ”the 10 most recent posts” doesn't need a sort at all. PostgreSQL reads the index backwards and stops after 10:

```
EXPLAIN ANALYZE SELECT slug FROM posts ORDER BY published_on DESC LIMIT 10;
```

```
 Limit  (cost=0.28..1.83 rows=10 width=26) (actual time=0.010..0.011 rows=10.00 loops=1)
   ->  Index Scan Backward using posts_published_on_idx on posts  (cost=0.28..339.83 rows=2193 width
     ↪ =26) (actual time=0.009..0.010 rows=10.00 loops=1)
         Index Searches: 1
 Execution Time: 0.015 ms
```

Look at the costs: the full scan would cost 339.83, but `Limit` knows it needs only 10 rows, so it costs 1.83.

When estimates go wrong

The planner bases everything on statistics that `ANALYZE` collects: how many rows, how many distinct values, the most common values. Autovacuum runs `ANALYZE` for you after enough rows change. You can see what it collected in `pg_stats`.

Statistics are kept per column. Wrap a column in a function, and the planner doesn't know what the result looks like:

```
EXPLAIN ANALYZE SELECT count(*) FROM posts WHERE extract(year FROM published_on) = 2026;
```

```
 Aggregate  (cost=288.92..288.93 rows=1 width=8) (actual time=0.459..0.459 rows=1.00 loops=1)
   ->  Seq Scan on posts  (cost=0.00..288.89 rows=11 width=0) (actual time=0.068..0.445 rows=509.00
    ↪ loops=1)
         Filter: (EXTRACT(year FROM published_on) = '2026'::numeric)
         Rows Removed by Filter: 1684
 Execution Time: 0.473 ms
```

Expected 11 rows, got 509. Off by a factor of 46. In a bigger query, an estimate like that makes the planner choose a nested loop where a hash join would be ten times faster.

There are two fixes. Write the condition on the plain column, which also lets an index on `published_on` work:

```
WHERE published_on >= '2026-01-01' AND published_on < '2027-01-01'
```

Or tell PostgreSQL to collect statistics on the expression:

```
CREATE STATISTICS posts_year_stats ON (extract(year FROM published_on)) FROM posts;
ANALYZE posts;
```

```
   ->  Seq Scan on posts  (cost=0.00..288.89 rows=509 width=0) (actual time=0.051..0.354 rows=509.00
    ↪ loops=1)
```

Now the estimate is exact: 509.

My checklist for a slow query

1. Run `EXPLAIN ANALYZE` on the real query, with real parameters, on real data. A plan from your laptop's 100 test rows tells you nothing about production's 5 million.
2. Find the node where the time goes. `actual time` includes the children, so subtract theirs.
3. Compare estimated and actual rows. A big difference points to missing or stale statistics. Run `ANALYZE` and look again.
4. Look for a `Seq Scan` with `Rows Removed by Filter` much bigger than the rows kept, on a big table. That's a missing index.
5. Look at `Buffers`. Lots of pages read for few rows means wasted work.

For long plans, explain.depesz.com colors the slow nodes for you. Paste the text output of `EXPLAIN ANALYZE`.

Indexes

A table with 2,193 rows is fast whatever you do. To see what indexes are worth, I need a bigger table. So I made one: 5 million page views, generated with random posts, times and countries. The views are not real, the posts they point to are:

```
CREATE TABLE post_views (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  post_id bigint NOT NULL REFERENCES posts (id),
  viewed_at timestamptz NOT NULL,
  country text NOT NULL
);

INSERT INTO post_views (post_id, viewed_at, country)
SELECT 1 + floor(random() * 2193)::int,
       timestamptz '2026-01-01' + random() * interval '330 days',
       (array['US','IN','DE','GB','IT','FR','BR','CA','ES','NL'])[1 + floor(random() * 10)::int]
FROM generate_series(1, 5000000);
ANALYZE post_views;
```

```
INSERT 0 5000000
Time: 22225.507 ms (00:22.226)
```

22 seconds, and the table takes 394 MB with its primary key. We'll come back to why the insert took that long.

Without an index

How many times was my PostgreSQL introduction viewed?

```
EXPLAIN ANALYZE SELECT count(*) FROM post_views WHERE post_id = 732;
```

```
 Finalize Aggregate  (cost=63809.19..63809.20 rows=1 width=8) (actual time=55.642..56.805 rows=1.00
    ↪ loops=1)
   Buffers: shared hit=15720 read=21045 written=94
   ->  Gather  (cost=63808.97..63809.18 rows=2 width=8) (actual time=55.566..56.800 rows=3.00 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=62808.97..62808.98 rows=1 width=8) (actual time=53.443..53.444
    ↪ rows=1.00 loops=3)
               ->  Parallel Seq Scan on post_views  (cost=0.00..62806.62 rows=941 width=0) (actual time
    ↪ =0.505..53.387 rows=759.33 loops=3)
                     Filter: (post_id = 732)
                     Rows Removed by Filter: 1665907
 Execution Time: 56.839 ms
```

Three processes (the main one and two workers) split the table and each read a third of it, about 1.67 million rows, to find 2,278 views. 57 ms and 36,765 pages.

A B-tree index

The default index type is the **B-tree**, a sorted tree that finds a value in a few steps, like a phone book:

```
\timing on
CREATE INDEX post_views_post_id_idx ON post_views (post_id);
```

```
CREATE INDEX
Time: 1421.772 ms (00:01.422)
```

```
 Aggregate  (cost=7222.12..7222.13 rows=1 width=8) (actual time=4.419..4.419 rows=1.00 loops=1)
   ->  Bitmap Heap Scan on post_views  (cost=25.94..7216.48 rows=2259 width=0) (actual time
     ↪ =0.302..4.357 rows=2278.00 loops=1)
         Recheck Cond: (post_id = 732)
         Heap Blocks: exact=2218
         ->  Bitmap Index Scan on post_views_post_id_idx  (cost=0.00..25.38 rows=2259 width=0) (actual
     ↪ time=0.130..0.130 rows=2278.00 loops=1)
               Index Cond: (post_id = 732)
               Index Searches: 1
 Execution Time: 4.447 ms
```

From 57 ms to 4.4 ms. The index found the 2,278 rows in 0.13 ms. The rest of the time went into reading 2,218 table pages, one per row, because random views of the same post are scattered all over the table.

Index your foreign keys

Remember that PostgreSQL doesn't index the referencing side of a foreign key? Here's why it matters. Deleting a post means checking that no view points to it. Before I created that index, the check had to scan `post_views`:

```
BEGIN;
DELETE FROM posts WHERE id = 5;
ROLLBACK;
```

```
ERROR:  update or delete on table "posts" violates foreign key constraint "post_views_post_id_fkey" on
    ↪ table "post_views"
DETAIL:  Key (id)=(5) is still referenced from table "post_views".
Time: 39.426 ms
```

With the index, the same check took 1.591 ms. Now picture a job that deletes a thousand posts: 39 seconds or 1.6.

The other direction has a cost too. Inserting a view means checking that the post exists, with a lookup in `posts_pkey`. That's one reason the 5 million inserts took 22 seconds, and it shows in the index statistics we'll look at later: `posts_pkey` was used 5,002,458 times.

Multicolumn indexes

Now let's ask for the views of one post since the start of November:

```
CREATE INDEX post_views_post_time_idx ON post_views (post_id, viewed_at);
```

```
Time: 3606.531 ms (00:03.607)
```

```
Aggregate  (cost=661.91..661.92 rows=1 width=8) (actual time=0.309..0.310 rows=1.00 loops=1)
  ->  Bitmap Heap Scan on post_views  (cost=6.20..661.48 rows=172 width=0) (actual time=0.039..0.302
    ↪ rows=174.00 loops=1)
        Recheck Cond: ((post_id = 732) AND (viewed_at >= '2026-11-01 00:00:00+01'::timestamp with time
    ↪  zone))
        Heap Blocks: exact=174
        ->  Bitmap Index Scan on post_views_post_time_idx  (cost=0.00..6.15 rows=172 width=0) (actual
    ↪ time=0.027..0.027 rows=174.00 loops=1)
              Index Cond: ((post_id = 732) AND (viewed_at >= '2026-11-01 00:00:00+01'::timestamp with
    ↪ time zone))
              Index Searches: 1
Execution Time: 0.335 ms
```

0.3 ms. The index is sorted by `post_id` first, then by `viewed_at` inside each post, so the 174 rows we want sit next to each other in the index.

The order of the columns matters. This index is perfect for "one post, a time range", and it also works for "one post" alone. For "a time range, any post", the time values are spread across 2,193 separate sections of the index, one per post.

Skip scan in PostgreSQL 18

Until PostgreSQL 17, an index on `(post_id, viewed_at)` was useless for a condition on `viewed_at` alone. PostgreSQL 18 adds **skip scan**: it jumps to each `post_id` in turn and searches the time range inside it. I dropped the single-column index on `post_id` and asked for one day of views across all posts:

```
EXPLAIN ANALYZE SELECT count(*) FROM post_views
WHERE viewed_at >= '2026-11-01' AND viewed_at < '2026-11-02';
```

```
Aggregate  (cost=37531.04..37531.05 rows=1 width=8) (actual time=66.889..66.890 rows=1.00 loops=1)
  ->  Bitmap Heap Scan on post_views  (cost=9410.18..37494.10 rows=14776 width=0) (actual time
    ↪ =9.725..66.392 rows=15231.00 loops=1)
        Recheck Cond: ((viewed_at >= '2026-11-01 00:00:00+01'::timestamp with time zone) AND (
    ↪ viewed_at < '2026-11-02 00:00:00+01'::timestamp with time zone))
        Heap Blocks: exact=12479
        ->  Bitmap Index Scan on post_views_post_time_idx  (cost=0.00..9406.49 rows=14776 width=0) (
    ↪ actual time=8.711..8.711 rows=15231.00 loops=1)
              Index Cond: ((viewed_at >= '2026-11-01 00:00:00+01'::timestamp with time zone) AND (
    ↪ viewed_at < '2026-11-02 00:00:00+01'::timestamp with time zone))
              Index Searches: 2195
Execution Time: 66.920 ms
```

`Index Searches: 2195` is the skip scan: one search per distinct `post_id`, plus a couple. The index part took 8.7 ms. But the whole query took 67 ms, against 61 ms for the parallel sequential scan it did before, because the 15,231 matching rows live on 12,479 different pages. Skip scan works best when the first column has few distinct values, and it doesn't replace an index designed for the query. An index on `viewed_at` would be the right fix here.

Index-only scans and the visibility map

When every column a query needs is in the index, PostgreSQL can skip the table entirely. After a `VACUUM`, the same "one post since November" query:

```
VACUUM post_views;
```

```
 Aggregate  (cost=8.30..8.31 rows=1 width=8) (actual time=0.069..0.069 rows=1.00 loops=1)
   ->  Index Only Scan using post_views_post_time_idx on post_views  (cost=0.43..7.87 rows=172 width=0)
     ↪  (actual time=0.048..0.057 rows=174.00 loops=1)
           Index Cond: ((post_id = 732) AND (viewed_at >= '2026-11-01 00:00:00+01'::timestamp with time
     ↪ zone))
           Heap Fetches: 0
           Index Searches: 1
 Execution Time: 0.091 ms
```

`Heap Fetches: 0`: not one table page read. 0.09 ms.

Why did it need the `VACUUM`? The index doesn't know whether a row is visible to your transaction (the MVCC chapter explains what that means). PostgreSQL keeps a **visibility map**, one bit per table page saying "every row here is visible to everyone". `VACUUM` sets those bits. For pages with the bit set, an index-only scan can trust the index. For the others it must check the table, and that's what `Heap Fetches` counts.

You saw it earlier with posts: `Heap Fetches: 509` for 509 rows, because I had just rewritten the table.

Partial indexes

A partial index only includes the rows that match a `WHERE`. If a dashboard always looks at views from Italy:

```
CREATE INDEX post_views_italy_idx ON post_views (viewed_at) WHERE country = 'IT';
```

```
 partial | full_idx
---------+----------
 11 MB   | 150 MB
(1 row)
```

11 MB against 150 MB for the two-column index, because it holds a tenth of the rows and one column. A query with the same condition uses it:

```
EXPLAIN ANALYZE SELECT count(*) FROM post_views WHERE country = 'IT' AND viewed_at >= '2026-11-01';
```

```
 Aggregate  (cost=1172.48..1172.49 rows=1 width=8) (actual time=2.735..2.735 rows=1.00 loops=1)
   ->  Index Only Scan using post_views_italy_idx on post_views  (cost=0.42..1077.97 rows=37803 width
     ↪ =0) (actual time=0.038..1.828 rows=39443.00 loops=1)
           Index Cond: (viewed_at >= '2026-11-01 00:00:00+01'::timestamp with time zone)
           Heap Fetches: 0
           Index Searches: 1
 Execution Time: 2.756 ms
```

Partial indexes are great for "active" rows: `WHERE deleted_at IS NULL`, `WHERE status = 'pending'`. And a partial *unique* index enforces rules like "one active subscription per user" while ignoring the canceled ones.

Expression indexes

An index on a column doesn't help a condition on a function of that column. Case-insensitive title lookup:

```
EXPLAIN ANALYZE SELECT id FROM posts WHERE lower(title) = 'introduction to postgresql';
```

```
 Seq Scan on posts  (cost=0.00..288.89 rows=11 width=8) (actual time=0.873..0.878 rows=1.00 loops=1)
   Filter: (lower(title) = 'introduction to postgresql'::text)
   Rows Removed by Filter: 2192
 Execution Time: 0.896 ms
```

Index the expression itself:

```
CREATE INDEX posts_lower_title_idx ON posts (lower(title));
ANALYZE posts;
```

```
 Index Scan using posts_lower_title_idx on posts  (cost=0.28..8.30 rows=1 width=8) (actual time
    ↪ =0.023..0.023 rows=1.00 loops=1)
   Index Cond: (lower(title) = 'introduction to postgresql'::text)
   Index Searches: 1
 Execution Time: 0.030 ms
```

The query must use the exact same expression. `lower(title)` matches, `upper(title)` doesn't.

Index types

Type	Good for	Example in this book
B-tree (default)	equality, ranges, sorting, prefix `LIKE 'abc%'`	`post_views (post_id, viewed_at)`
GIN	values with many parts: JSONB, arrays, `tsvector`	the frontmatter and search indexes
GiST	ranges, geometry, ”overlaps” and ”nearest”	the bookings exclusion constraint
BRIN	huge tables physically sorted by a column, like append-only logs by time	
Hash	equality only	rarely better than B-tree

CREATE INDEX CONCURRENTLY

A plain `CREATE INDEX` blocks every write to the table while it builds. On my 5 million rows that's a few seconds. On a production table with a billion rows, it's an outage.

`CONCURRENTLY` builds the index without blocking writes:

```
CREATE INDEX CONCURRENTLY post_views_country_idx ON post_views (country);
```

```
CREATE INDEX
Time: 2478.818 ms (00:02.479)
```

It's slower, because it scans the table twice, and it has rules. It can't run inside a transaction:

```
ERROR:  CREATE INDEX CONCURRENTLY cannot run inside a transaction block
```

That matters for migration tools that wrap every migration in a transaction. And if it fails halfway (a unique index finding a duplicate, say), it leaves an `INVALID` index behind that you need to drop and create again. `\d post_views` shows invalid indexes with the word `INVALID`.

On a live database, I always use `CONCURRENTLY`.

Indexes are not free

Every index is a copy of some data, kept up to date on every write. Two measurements on my table.

Size. The indexes on `post_views` take more space than the table itself:

```
SELECT pg_size_pretty(pg_relation_size('post_views')) AS heap,
       pg_size_pretty(pg_indexes_size('post_views')) AS indexes;
```

```
  heap  | indexes
--------+---------
 287 MB | 301 MB
(1 row)
```

Write speed. I inserted the same million rows into two copies of the table, one without indexes and one with four:

```
INSERT 0 1000000
Time: 1415.745 ms (00:01.416)
...
INSERT 0 1000000
Time: 7661.329 ms (00:07.661)
```

1.4 seconds against 7.7. Five times slower.

Finding unused indexes

PostgreSQL counts how many times each index is used. After all the experiments in this chapter:

```
SELECT indexrelname, idx_scan, pg_size_pretty(pg_relation_size(indexrelid))
FROM pg_stat_user_indexes ORDER BY idx_scan NULLS FIRST, 1;
```

```
       indexrelname       | idx_scan | pg_size_pretty
--------------------------+----------+----------------
 post_views_country_idx   |        0 | 33 MB
 post_views_pkey          |        0 | 107 MB
 post_views_italy_idx     |        1 | 11 MB
 posts_lower_title_idx    |        1 | 144 kB
 posts_frontmatter_gin    |        2 | 424 kB
 page_views_pkey          |        9 | 16 kB
 posts_search_idx         |       10 | 6288 kB
 post_tags_pkey           |     2048 | 88 kB
 post_views_post_time_idx |     2197 | 150 MB
 tags_name_key            |     2240 | 16 kB
 posts_slug_key           |     2273 | 112 kB
 tags_pkey                |     2289 | 16 kB
 posts_pkey               |  5002458 | 64 kB
(13 rows)
```

`post_views_country_idx` has never been used: 33 MB of disk, and slower inserts, for nothing. On a production database that's been running for weeks, an index with `idx_scan = 0` is a candidate for `DROP INDEX CONCURRENTLY`. Keep the ones that enforce something, like `post_views_pkey`, even if no query uses them.

And there's `posts_pkey` with 5 million uses: the foreign key checks from the inserts into `post_views`.

Transactions, MVCC and locks

The SQL Handbook covered `BEGIN`, `COMMIT` and `ROLLBACK`. This chapter is about what happens when several transactions run at the same time, which is where PostgreSQL is most interesting, and where most production surprises come from.

To show it, I ran two or three `psql` sessions at the same time, with `pg_sleep` to control the timing, and recorded what each one printed.

MVCC: rows have versions

PostgreSQL never updates a row in place. An `UPDATE` writes a new version of the row and marks the old one as expired. This is called **multiversion concurrency control**, or MVCC.

You can see it with hidden system columns. `ctid` is the physical location of a row version (page, item), `xmin` is the transaction that created it, `xmax` the one that expired it:

```
CREATE TABLE counters (name text PRIMARY KEY, value int NOT NULL);
INSERT INTO counters VALUES ('newsletter', 100), ('downloads', 0);
SELECT ctid, xmin, xmax, * FROM counters;
```

```
 ctid  | xmin | xmax |    name    | value
-------+------+------+------------+-------
 (0,1) |  928 |    0 | newsletter |   100
```

```
 (0,2) |  928 |    0 | downloads  |     0
(2 rows)
```

Both rows were created by transaction 928, and nobody has expired them. Now an update:

```
UPDATE counters SET value = value + 1 WHERE name = 'newsletter';
SELECT ctid, xmin, xmax, * FROM counters;
```

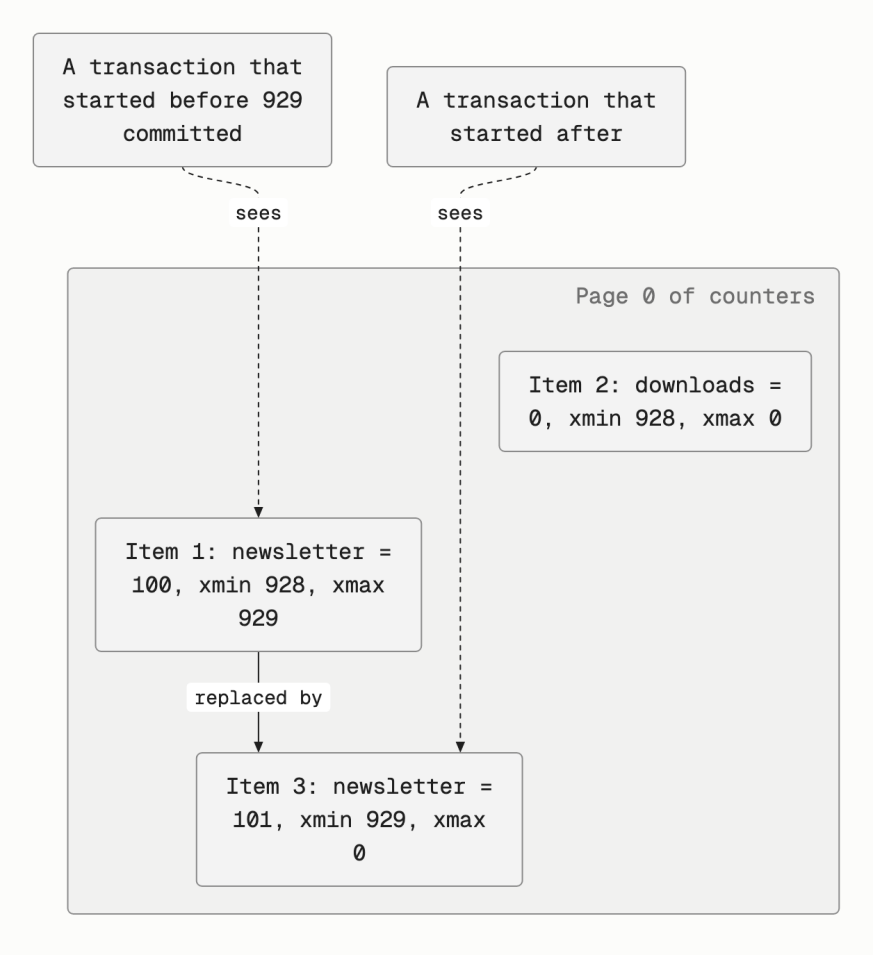
```
 ctid  | xmin | xmax |    name    | value
-------+------+------+------------+-------
 (0,2) |  928 |    0 | downloads  |     0
 (0,3) |  929 |    0 | newsletter |   101
(2 rows)
```

The newsletter row moved to `(0,3)`, created by transaction 929. Where did `(0,1)` go? It's still in the page. The `pageinspect` extension shows the raw contents:

```
CREATE EXTENSION pageinspect;
SELECT lp, t_xmin, t_xmax, t_ctid FROM heap_page_items(get_raw_page('counters', 0));
```

```
 lp | t_xmin | t_xmax | t_ctid
----+--------+--------+--------
  1 |    928 |    929 | (0,3)
  2 |    928 |      0 | (0,2)
  3 |    929 |      0 | (0,3)
(3 rows)
```

The old version is there, with `xmax = 929` (expired by transaction 929) and a pointer to its replacement at `(0,3)`.



Each transaction sees the row versions that were committed when its snapshot was taken, and ignores the rest. A reader looking at the old version and a writer creating the new one never wait for each other.

The price is the old versions. Somebody has to remove them once no transaction can see them anymore. That's `VACUUM`, at the end of this chapter.

Read committed

The default isolation level is **read committed**: each statement sees what was committed when the statement started. Two statements in the same transaction can see different data.

Session A reads the counter twice, two seconds apart. In between, session B sets it to 200:

```
--- session A
BEGIN
     step     | value
--------------+-------
 A first read |   101
(1 row)

     step      | value
---------------+-------
```

```
 A second read |   200
(1 row)

COMMIT
--- session B
UPDATE 1
```

Same transaction, same query, different answers. Usually that's fine. When it isn't, there's repeatable read.

Repeatable read

At **repeatable read**, the whole transaction sees one snapshot, taken at its first query. I reset the counter to 100 and ran the same two sessions, with A also updating the counter at the end:

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT 'A first read' AS step, value FROM counters WHERE name = 'newsletter';
SELECT pg_sleep(2);
SELECT 'A second read' AS step, value FROM counters WHERE name = 'newsletter';
UPDATE counters SET value = value + 1 WHERE name = 'newsletter';
COMMIT;
```

```
--- session A
BEGIN
     step     | value
--------------+-------
 A first read |   100
(1 row)

     step      | value
---------------+-------
 A second read |   100
(1 row)

ERROR:  could not serialize access due to concurrent update
ROLLBACK
--- session B
UPDATE 1
```

A saw 100 both times, even though B had committed 200 in between. Then A tried to update a row that had changed since its snapshot, and PostgreSQL refused: A would be writing based on a stale view. The application has to retry the whole transaction.

The lost update

Here's a classic bug. Two requests increment a counter by reading it, adding one in the application, and writing it back:

```
BEGIN;
SELECT value AS read_value FROM counters WHERE name = 'newsletter' \gset
SELECT pg_sleep(1);
UPDATE counters SET value = :read_value + 1 WHERE name = 'newsletter';
COMMIT;
```

I ran it in two sessions at the same time, starting from 100:

```
 after_two_increments
----------------------
                   101
(1 row)
```

Two increments, and the counter went up by one. Both sessions read 100, both wrote 101. That's a **lost update**, and read committed doesn't prevent it.

The fix is to do the arithmetic in the database, in one statement:

```
UPDATE counters SET value = value + 1 WHERE name = 'newsletter';
```

Same two concurrent sessions:

```
 after_two_increments
----------------------
                   102
(1 row)
```

The second `UPDATE` waits for the first to commit, then re-reads the row and adds one to the new value. When the logic is too complex for one statement, lock the row first with `SELECT ... FOR UPDATE`, which we'll see in a moment.

Serializable

Serializable is the strictest level: PostgreSQL guarantees the result is the same as if the transactions had run one at a time, in some order, and aborts one when it can't.

The classic example is the on-call rota. Two doctors are on call, and the rule is that at least one must stay. Each doctor's transaction checks that there are two on call, then takes itself off:

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT count(*) AS on_duty FROM on_call WHERE on_duty;
SELECT pg_sleep(1);
UPDATE on_call SET on_duty = false WHERE doctor = 'alice';
COMMIT;
```

Alice and Bob run it at the same time:

```
--- alice
 on_duty
```

```
---------
       2
(1 row)

UPDATE 1
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
--- bob
 on_duty
---------
       2
(1 row)

UPDATE 1
COMMIT
```

Bob's commit went through, Alice's failed at `COMMIT`. If she retries, she'll see only one doctor on call and stay.

The same two transactions at read committed both succeed, and the rule is broken:

```
 doctor | on_duty
--------+---------
 alice  | f
 bob    | f
(2 rows)
```

Nobody is on call. Each transaction updated a *different* row, so no lock stopped them, and each one made its decision on a count that was true when it read it. This is called **write skew**, and only serializable catches it.

(My first run of this test showed both doctors still on duty. I had passed the doctor's name as a `psql` variable inside `-c`, where `psql` doesn't replace variables, so both `UPDATE`s failed with a syntax error that went to `/dev/null`. Concurrent tests are easy to get wrong. Always check the output of every session.)

Level	Each statement sees	Prevents lost updates	Needs retries
Read committed (default)	data committed before the statement	only with atomic updates or `FOR UPDATE`	no
Repeatable read	one snapshot for the whole transaction	yes, by aborting	yes
Serializable	one snapshot, plus conflict detection	yes, including write skew	yes

My default: read committed, with atomic `UPDATE`s and explicit row locks where needed. Serializable when the rules span several rows and I can retry the transaction in code.

Row locks: who is blocking whom

Readers never wait for writers. Writers do wait for other writers on the same row. Session A updates the downloads counter and keeps its transaction open for 4 seconds. Session B tries to update the same row:



From a third session, `pg_stat_activity` shows what each connection is doing:

```
SELECT pid, state, wait_event_type, wait_event, left(query, 60) AS query
FROM pg_stat_activity
WHERE datname = 'blog' AND pid <> pg_backend_pid() ORDER BY pid;
```

```
  pid  | state  | wait_event_type |  wait_event   |                            query
-------+--------+-----------------+---------------+-----------------------------------------------------------
 11808 | active | Timeout         | PgSleep       | SELECT pg_sleep(4);
 11819 | active | Lock            | transactionid | update counters set value = value + 10 where name =
    ↪  'downloa
(2 rows)
```

And `pg_blocking_pids` says who is in the way:

```
SELECT pid, pg_blocking_pids(pid) AS blocked_by, left(query, 50) AS query
FROM pg_stat_activity WHERE cardinality(pg_blocking_pids(pid)) > 0;
```

```
  pid  | blocked_by |                       query
-------+------------+-------------------------------------------------
 11819 | {11808}    | update counters set value = value + 10 where name
(1 row)
```

Session B waited 3.5 seconds and then went through, as soon as A committed:

```
UPDATE 1
B waited: 3.509804000 s
```

Note `wait_event` = `transactionid`: B isn't waiting for the row as such, it's waiting for A's transaction to finish, because only then will it know whether A's version of the row is the one to update. `pg_locks` shows that detail: A holds an `ExclusiveLock` on its own transaction ID, and B wants a `ShareLock` on it, not granted.

This is the query to run first when a production database ”hangs”. Very often the cause is one transaction that someone left open.

If you have the PID of a session you want to stop, `SELECT pg_cancel_backend(11808)` cancels its current query, and `pg_terminate_backend(11808)` closes the connection.

lock_timeout

A statement that waits for a lock waits forever by default. `lock_timeout` sets a limit:

```
SET lock_timeout = '1s';
UPDATE counters SET value = value + 10 WHERE name = 'downloads';


ERROR:  canceling statement due to lock timeout
CONTEXT:  while updating tuple (0,14) in relation "counters"
```

Failing fast is often better than piling up waiting requests. It's essential for migrations, as we'll see.

Deadlocks

A deadlock is two transactions each waiting for a lock the other holds. A updates the newsletter row, then the downloads row. B does the same in the opposite order. Both start at the same time:

```
--- session A
BEGIN
UPDATE 1
ERROR:  deadlock detected
DETAIL:  Process 11977 waits for ShareLock on transaction 946; blocked by process 11978.
Process 11978 waits for ShareLock on transaction 945; blocked by process 11977.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,15) in relation "counters"
ROLLBACK
--- session B
BEGIN
UPDATE 1
UPDATE 1
COMMIT
```

PostgreSQL checks for deadlocks after a waiting lock has waited for one second (`deadlock_timeout`), picks a victim, and aborts it so the other can go on. The server log has the full story, including both queries:

```
ERROR:  deadlock detected
DETAIL:  Process 11977 waits for ShareLock on transaction 946; blocked by process 11978.
    Process 11978 waits for ShareLock on transaction 945; blocked by process 11977.
    Process 11977: UPDATE counters SET value = value + 1 WHERE name = 'downloads';
    Process 11978: UPDATE counters SET value = value + 1 WHERE name = 'newsletter';
```

The fix is in the application: always lock rows in the same order (for example by primary key), and retry the transaction when you get a deadlock error.

SELECT FOR UPDATE

When you need to read a row, decide something in code, and then write it, lock the row as you read it. `SELECT ... FOR UPDATE` takes the same lock an `UPDATE` would:

```
BEGIN;
SELECT value FROM counters WHERE name = 'downloads' FOR UPDATE;
-- decide in the application, then:
UPDATE counters SET value = value + 1 WHERE name = 'downloads';
COMMIT;
```

Another session doing the same `SELECT ... FOR UPDATE` waits. Two variants change that. `NOWAIT` fails immediately:

```
ERROR:  could not obtain lock on row in relation "counters"
```

And `SKIP LOCKED` skips rows that are locked, returning the rest:

```
 value
-------
(0 rows)
```

`SKIP LOCKED` is how you build a job queue on PostgreSQL. Each worker runs `SELECT ... FROM jobs WHERE status = 'pending' ORDER BY id LIMIT 1 FOR UPDATE SKIP LOCKED`, and two workers never get the same job.

Dead rows and VACUUM

Back to the old row versions. I updated a million rows of `post_views`, without changing anything:

```
UPDATE post_views SET country = country WHERE id <= 1000000;
SELECT n_live_tup, n_dead_tup, last_autovacuum FROM pg_stat_user_tables WHERE relname = 'post_views';
```

```
 n_live_tup | n_dead_tup | last_autovacuum
------------+------------+-----------------
    5000000 |    1000000 |
(1 row)
```

A million **dead rows**: old versions nobody can see anymore, taking space and slowing down scans. `VACUUM` removes them and marks the space as reusable. With `VERBOSE` it explains what it did:

```
VACUUM (VERBOSE) post_views;
```

```
INFO:  vacuuming "blog.public.post_views"
INFO:  launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO:  finished vacuuming "blog.public.post_views": index scans: 1
pages: 0 removed, 44118 remain, 14707 scanned (33.34% of total), 0 eagerly scanned
tuples: 1000000 removed, 4999968 remain, 0 are dead but not yet removable
```

```
removable cutoff: 957, which was 0 XIDs old when operation ended
frozen: 6275 pages from table (14.22% of total) had 853400 tuples frozen
visibility map: 14707 pages set all-visible, 13627 pages set all-frozen (0 were all-visible)
index scan needed: 7353 pages from table (16.67% of total) had 1000000 dead item identifiers removed
...
system usage: CPU: user: 0.24 s, system: 0.07 s, elapsed: 0.44 s
```

A million dead rows removed in 0.44 seconds, and the visibility map updated (that's what made the index-only scan possible). Note "0 are dead but not yet removable": if a transaction had been open since before my `UPDATE`, those rows would still be visible to it, and `VACUUM` couldn't remove them.

`VACUUM` doesn't give the space back to the operating system. The table went from 287 MB to 345 MB with the update, and stays 345 MB, with the free space reused by future writes. `VACUUM FULL` rewrites the table to its minimal size, but it locks the table completely while it does, so it's not something to run on a live database.

Autovacuum

You rarely run `VACUUM` by hand, because **autovacuum** does it in the background. It vacuums a table when its dead rows pass a threshold:

```
              name                |  setting
----------------------------------+-----------
 autovacuum                       | on
 autovacuum_max_workers           | 3
 autovacuum_naptime               | 60
 autovacuum_vacuum_max_threshold  | 100000000
 autovacuum_vacuum_scale_factor   | 0.2
 autovacuum_vacuum_threshold      | 50
 autovacuum_worker_slots          | 16
```

The rule is 50 dead rows plus 20% of the table. On a table with 5 million rows, autovacuum waits for a million dead ones. PostgreSQL 18 adds `autovacuum_vacuum_max_threshold`, a cap of 100 million so huge tables don't wait forever, and `autovacuum_worker_slots`, which lets you change the number of workers without a restart.

For busy big tables, lower the scale factor on that table only:

```
ALTER TABLE post_views SET (autovacuum_vacuum_scale_factor = 0.02);
```

The one thing that breaks autovacuum is a long-running transaction. As long as it's open, no row version that it could see can be removed, in any table. An `idle in transaction` session left open for a day can make every table in the database grow. We'll catch one in the health chapter.

Node.js with pg

Time to use the database from an application. We'll build a small search API over the posts in Node.js, with `pg`, the most used PostgreSQL driver for Node. It connects

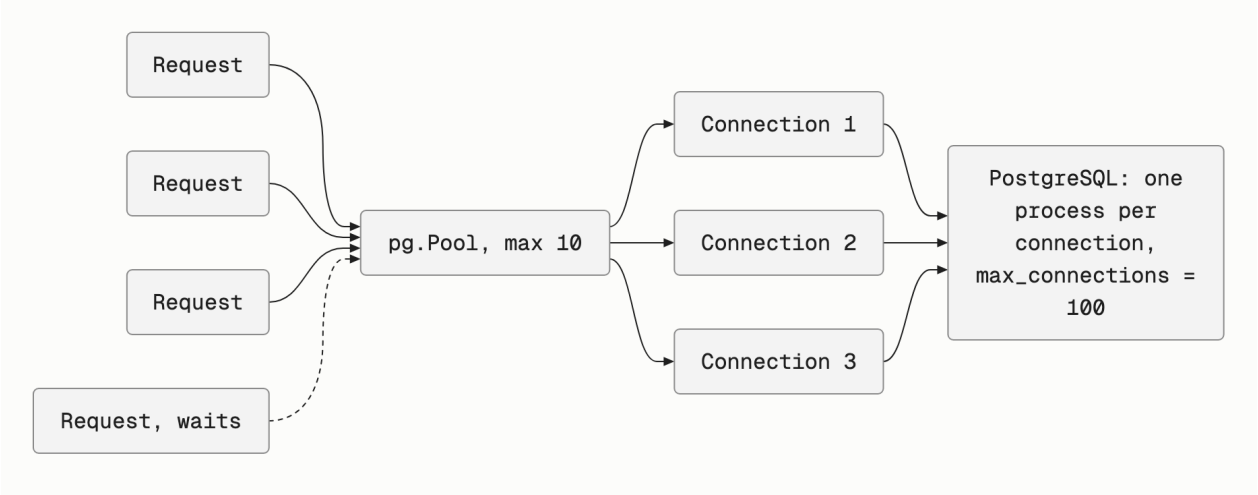
as `blog_app`, the restricted role from the permissions chapter.

```
npm install pg
```

Why a pool

Every PostgreSQL connection is a server process, and opening one takes a few milliseconds of authentication and setup. An application that opened a new connection for every request would spend more time connecting than querying, and a traffic spike would mean hundreds of server processes.

A **pool** opens a few connections and lends them to requests. A request borrows a connection, runs its queries, and gives it back.



The total across all your app instances must stay under the server's `max_connections`, which is 100 by default, minus a few kept for superusers. Four instances with a pool of 10 each use 40.

The server

Here's the whole thing:

```
// A small search API over the blog database.
//   DATABASE_URL=postgresql://blog_app:...@127.0.0.1:5418/blog node server.js
import http from 'node:http'
import pg from 'pg'

// Return DATE columns as 'YYYY-MM-DD' strings instead of JavaScript Dates
// at local midnight, which JSON.stringify shifts to the previous day in UTC.
pg.types.setTypeParser(pg.types.builtins.DATE, (value) => value)

const pool = new pg.Pool({
  connectionString: process.env.DATABASE_URL,
  max: 10,
  idleTimeoutMillis: 30_000,
  connectionTimeoutMillis: 2_000,
  application_name: 'blog-search',
})

// An idle client that loses its connection emits 'error' on the pool.
// Without a listener, that error crashes the process.
pool.on('error', (err) => console.error('idle client error:', err.message))
```

```
async function search(text, limit) {
  const { rows } = await pool.query(
    `SELECT slug, title, published_on,
            round(ts_rank(search, query)::numeric, 3) AS rank
     FROM posts, websearch_to_tsquery('english', $1) AS query
     WHERE search @@ query
     ORDER BY rank DESC, published_on DESC
     LIMIT $2`,
    [text, limit],
  )
  return rows
}

async function tagPost(slug, tagName) {
  const client = await pool.connect()
  try {
    await client.query('BEGIN')
    const tag = await client.query(
      `INSERT INTO tags (name) VALUES ($1)
       ON CONFLICT (name) DO UPDATE SET name = excluded.name
       RETURNING id`,
      [tagName],
    )
    const link = await client.query(
      `INSERT INTO post_tags (post_id, tag_id)
       SELECT id, $2 FROM posts WHERE slug = $1
       ON CONFLICT DO NOTHING`,
      [slug, tag.rows[0].id],
    )
    if (link.rowCount === 0) throw new Error(`no post "${slug}", or already tagged`)
    await client.query('COMMIT')
    return { slug, tag: tagName }
  } catch (err) {
    await client.query('ROLLBACK')
    throw err
  } finally {
    client.release()
  }
}

function send(res, status, body) {
  res.writeHead(status, { 'content-type': 'application/json' })
  res.end(JSON.stringify(body, null, 2) + '\n')
}

const server = http.createServer(async (req, res) => {
  const url = new URL(req.url, 'http://localhost')
  try {
    if (req.method === 'GET' && url.pathname === '/search') {
      const q = url.searchParams.get('q') ?? ''
      const limit = Math.min(Number(url.searchParams.get('limit')) || 5, 50)
      return send(res, 200, await search(q, limit))
    }
    if (req.method === 'POST' && url.pathname === '/tag') {
      const slug = url.searchParams.get('slug')
      const tag = url.searchParams.get('tag')
      return send(res, 200, await tagPost(slug, tag))
    }
    send(res, 404, { error: 'not found' })
  } catch (err) {
    console.error(err.message)
    send(res, 500, { error: err.message })
  }
})

server.listen(3000, () => console.log('listening on http://localhost:3000'))

process.on('SIGTERM', () => {
```

```
  server.close(async () => {
    await pool.end()
    console.log('pool closed, bye')
    process.exit(0)
  })
})
```

Let's go through the parts that matter.

pool.query for one statement

`pool.query()` borrows a connection, runs one query, and returns it, all in one call. Use it whenever a request runs a single statement, like `search()`.

The values go in a separate array, and the query refers to them as `$1`, `$2`. The driver sends the query text and the values separately, and the server never parses the values as SQL. That's what makes SQL injection impossible. Here's the difference, in a test script:

```
const input = "x' OR '1'='1"

const bad = await pool.query(`SELECT count(*) FROM posts WHERE slug = '${input}'`)
console.log('interpolated:', bad.rows[0].count)

const good = await pool.query('SELECT count(*) FROM posts WHERE slug = $1', [input])
console.log('parameter:   ', good.rows[0].count)
```

```
interpolated: 2193
parameter:    0
```

The interpolated query matched every post, because the input rewrote the `WHERE`. The parameterized one looked for a post with that strange slug, and found none. The SQL Handbook has a full chapter on injection, with an attack that drops a table through `pg`.

Parameters are also typed by the server, so bad input fails cleanly:

```
typed error:  invalid input syntax for type bigint: "abc"
```

pool.connect for a transaction

A transaction needs all its statements on the *same* connection. `pool.query()` might use a different one each time, so for `tagPost()` I take a client with `pool.connect()` and keep it until the end.

The structure is always the same: `BEGIN`, the work, `COMMIT`. On any error, `ROLLBACK`. And in `finally`, `client.release()`, whatever happened. Forget the release and the connection never goes back to the pool. After ten forgotten releases, the pool is empty and every request waits forever.

`tagPost()` creates the tag if it doesn't exist (the `DO UPDATE SET name = excluded.name` is a trick so `RETURNING` returns the ID even when the tag exists), then links it

to the post. If the post doesn't exist, or already has the tag, the link inserts no row, I throw, and the `ROLLBACK` removes the tag I may have just created.

Trying it

I start the server with the `blog_app` credentials:

```
DATABASE_URL='postgresql://blog_app:app-secret-123@127.0.0.1:5418/blog' node server.js
```

A search, using the trick from the full-text chapter to match both names:

```
curl 'http://localhost:3000/search?q=postgres%20or%20postgresql%20index&limit=3'
```

```
[
  {
    "slug": "connect-to-postgres-local-vs-vercel-postgres-with-kysely",
    "title": "Local PostgreSQL vs Vercel Postgres (now Neon) with Kysely",
    "published_on": "2023-11-10",
    "rank": "0.491"
  },
  {
    "slug": "cloudflare-hyperdrive",
    "title": "Cloudflare Hyperdrive tutorial: use PostgreSQL with Workers",
    "published_on": "2026-10-25",
    "rank": "0.474"
  },
  {
    "slug": "database-hosting-options",
    "title": "PostgreSQL and SQLite hosting: free and paid options",
    "published_on": "2026-09-27",
    "rank": "0.474"
  }
]
```

Tagging a post, then tagging it again, then tagging a post that doesn't exist:

```
curl -X POST 'http://localhost:3000/tag?slug=postgres-introduction&tag=postgresql'
curl -X POST 'http://localhost:3000/tag?slug=postgres-introduction&tag=postgresql'
curl -X POST 'http://localhost:3000/tag?slug=nope&tag=postgresql'
```

```
{
  "slug": "postgres-introduction",
  "tag": "postgresql"
}
{
  "error": "no post \"postgres-introduction\", or already tagged"
}
{
  "error": "no post \"nope\", or already tagged"
}
```

While the server runs, `pg_stat_activity` shows its connection, thanks to `application_name`:

```
  pid  | usename  | application_name | state |    q
-------+----------+------------------+-------+----------
 12443 | blog_app | blog-search      | idle  | ROLLBACK
(1 row)
```

One connection, idle, and its last statement was the `ROLLBACK` of the failed tag. Setting `application_name` costs nothing and saves you a lot of guessing when ten services share a database.

The date that moved back a day

My first version of the server didn't have the `setTypeParser` line, and the first search result said:

```
"published_on": "2023-11-09T23:00:00.000Z",
```

The post was published on November 10, 2023. Here's what happened. `pg` turns a `date` column into a JavaScript `Date` at midnight *in the time zone of the Node process*, which on my Mac is Rome. Midnight in Rome on November 10 is 23:00 UTC on November 9, and `JSON.stringify` writes dates in UTC. So every date in the API was one day early, for anyone east of Greenwich.

Telling `pg` to leave `date` values as strings fixes it:

```
pg.types.setTypeParser(pg.types.builtins.DATE, (value) => value)
```

```
default: 2023-11-09T23:00:00.000Z | text: 2023-11-10 | TZ Europe/Rome
parser: 2023-11-10
```

While we're on types: `bigint` and `numeric` come back as strings too, because they can be bigger than what a JavaScript number holds exactly. `count(*)` is a `bigint`:

```
bigint: { count: '2193', sum: '2088507' } string
```

Cast in SQL (`count(*)::int`) when you know the value is small, or convert in JavaScript.

When the pool runs out

What happens when more queries want a connection than the pool has? A test with a pool of 2, a 1-second wait for a free connection, and four queries that each take 1.5 seconds:

```
const pool = new pg.Pool({
  connectionString: process.env.DATABASE_URL,
```

```
  max: 2,
  connectionTimeoutMillis: 1_000,
})

const started = Date.now()
const results = await Promise.allSettled(
  [1, 2, 3, 4].map(async (n) => {
    await pool.query('SELECT pg_sleep(1.5)')
    return `query ${n} done after ${Date.now() - started} ms`
  }),
)
for (const r of results) console.log(r.status, r.value ?? r.reason.message)
```

```
fulfilled query 1 done after 1514 ms
fulfilled query 2 done after 1515 ms
rejected timeout exceeded when trying to connect
rejected timeout exceeded when trying to connect
```

Queries 3 and 4 waited one second for a connection and gave up. Without `connectionTimeoutMillis`, the default is to wait forever, and under load your requests pile up instead of failing. I always set it.

If your queries are fast, a small pool goes a long way: 10 connections running 5 ms queries serve 2,000 queries a second. A bigger pool rarely helps, and it can make things worse by overloading the server. When you have many app instances, or serverless functions that each open their own connections, put PgBouncer (or your host's pooler) between them and PostgreSQL.

The timeouts from the role

Remember the settings on the `blog_app` role? The permission error and the statement timeout both reach the application as errors with a code:

```
// as blog_reader
pool.query('insert into tags(name) values ($1)', ['x'])
// as blog_app
pool.query('select pg_sleep(6)')
```

```
42501 permission denied for table tags
57014 canceling statement due to statement timeout
```

`err.code` is the SQLSTATE code. These are the ones worth handling in code:

Code	Meaning	What to do
23505	unique violation	tell the user the value is taken
23503	foreign key violation	the referenced row doesn't exist
40001	serialization failure	retry the transaction
40P01	deadlock detected	retry the transaction

Code	Meaning	What to do
`57014`	statement timeout or cancel	the query was too slow
`42501`	insufficient privilege	a bug in your grants

Shutting down cleanly

When the process gets `SIGTERM` (which is what Docker, systemd and most platforms send before stopping it), the server stops accepting requests, waits for the ones in progress, then closes the pool:

```
pool closed, bye
```

Without `pool.end()`, the server sees connections drop without a goodbye. It cleans them up, but a transaction in the middle of a request would be rolled back without anyone knowing.

Migrations without downtime

A migration changes the schema of a database that's in use. On a small table almost anything works. On a big table some changes take seconds or minutes, and if they lock the table while they run, the application stops.

I measured the common changes on the 5-million-row `post_views` table.

Adding a column

Adding a column with a constant default is instant:

```
ALTER TABLE post_views ADD COLUMN source text NOT NULL DEFAULT 'web';


ALTER TABLE
Time: 1.612 ms
```

PostgreSQL stores the default once, in the catalog, and returns it for every old row without touching them. But only if the default is a constant. A default that's computed per row, like a new UUID, rewrites the whole table:

```
ALTER TABLE post_views ADD COLUMN import_id uuid DEFAULT uuidv7();


ALTER TABLE
Time: 9360.548 ms (00:09.361)
```

9.4 seconds, during which nobody could read or write the table.

Changing a column type

Most type changes rewrite the table too:

```
ALTER TABLE post_views ALTER COLUMN country TYPE varchar(2);


Time: 8547.872 ms (00:08.548)
```

Making the limit bigger, `varchar(2)` to `varchar(10)`, took 1.4 seconds, and going to `text` 0.2 seconds, because PostgreSQL knows those don't change the stored data. This is why I prefer `text` with a `CHECK` constraint: changing a check never touches the rows.

NOT NULL on an existing column

`SET NOT NULL` checks every row while holding the strongest lock:

```
ALTER TABLE post_views ALTER COLUMN source SET NOT NULL;


Time: 222.821 ms
```

0.2 seconds here, with the table in memory. On a table that's 100 times bigger and not cached, it's minutes of a locked table. The safe way takes three steps: add a check constraint as `NOT VALID` (instant), validate it (it scans, but with a lock that lets reads and writes go on), then `SET NOT NULL`, which sees the valid constraint and skips the scan:

```
ALTER TABLE post_views ADD CONSTRAINT source_nn CHECK (source IS NOT NULL) NOT VALID;
ALTER TABLE post_views VALIDATE CONSTRAINT source_nn;
ALTER TABLE post_views ALTER COLUMN source SET NOT NULL;
ALTER TABLE post_views DROP CONSTRAINT source_nn;


Time: 0.213 ms
Time: 130.704 ms
Time: 0.191 ms
Time: 0.102 ms
```

The only slow step is the validation, and it doesn't block the application. With PostgreSQL 18, `ADD CONSTRAINT ... NOT NULL ... NOT VALID` lets you skip the temporary check constraint.

The lock queue

Here's the problem that surprises people the most. My migration adds a column, which takes 2 milliseconds. But a long query is running on the table. Three sessions:

1. a report that reads `post_views` inside a transaction, still open
2. the migration: `ALTER TABLE post_views ADD COLUMN note text`

3. a normal, fast query from the application

```
  pid  | wait_event_type | wait_event | blocked_by |                          query
-------+-----------------+------------+------------+-------------------------------------------------
 13156 | Timeout         | PgSleep    | {}         | select pg_sleep(4)
 13211 | Lock            | relation   | {13156}    | alter table post_views add column note text
 13227 | Lock            | relation   | {13211}    | select count(*) from posts p join post_views v on
    ↪ v.pos
(3 rows)
```

The `ALTER TABLE` needs an exclusive lock, so it waits for the report. Fair enough. But the application query is blocked by the `ALTER TABLE`, not by the report. Locks are granted in order, and the fast query queued up behind the migration:

```
simple read waited 3.012778000 s
```

A 2-millisecond migration stopped every query on the table for 3 seconds. With a report that runs for 10 minutes, it's a 10-minute outage.

The fix is `lock_timeout`. If the migration can't get its lock quickly, it gives up, and you retry it later:

```
SET lock_timeout = '2s';
ALTER TABLE post_views ADD COLUMN note text;


SET
ERROR:  canceling statement due to lock timeout
```

I put `SET lock_timeout` at the top of every migration that touches a big table, with a retry loop around it.

My rules for migrations

Change	Safe way
Add a column	constant default only, or no default and fill it in batches
Add `NOT NULL`	`CHECK ... NOT VALID`, `VALIDATE`, then `SET NOT NULL` (or `NOT NULL ... NOT VALID` on 18)
Add a foreign key	`ADD CONSTRAINT ... NOT VALID`, then `VALIDATE CONSTRAINT`
Add an index	`CREATE INDEX CONCURRENTLY`, outside a transaction
Change a column type	add a new column, copy in batches, switch the code, drop the old one
Rename a column or table	add the new name, deploy code that uses it, then drop the old one
Anything else	`SET lock_timeout` first

Renaming is instant for the database, but not for your running code. When I renamed `page_views` to `post_page_views`, every query using the old name started failing that very second, until the new version of the app was deployed. Change the database and the code in steps, so each version of the code works with each state of the database.

Backups and restores

A backup you haven't restored is a hope, not a backup. In this chapter I back up the blog database in three formats, restore it into a new database, and check it.

pg_dump

`pg_dump` exports one database while it's in use. It runs inside a single repeatable read transaction, so the dump is consistent even if writes happen while it runs, and it doesn't block them.

The **plain** format is a SQL script. Here are just the post tables:

```
pg_dump -d blog -t posts -t tags -t post_tags -f backups/blog-core.sql
```

It took 0.4 seconds and wrote 17 MB. The top of the file:

```
--
-- PostgreSQL database dump
--

\restrict 6UYd4q7O9w9iLkitEXivkvsUVJIxd6bSmKCR7Eu43Pt1w9l6JwcUaD0fDf77Qld

-- Dumped from database version 18.7 (Homebrew)
-- Dumped by pg_dump version 18.7 (Homebrew)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
```

The `\restrict` line with a random key is recent: it stops a malicious dump from running `psql` meta-commands when you restore it. You restore a plain dump with `psql -f`.

The **custom** format (`-Fc`) is compressed, and `pg_restore` can pick what to restore from it:

```
pg_dump -d blog -Fc -f backups/blog.dump
```

7.3 seconds and 87 MB for the whole database, `post_views` included. The **directory** format (`-Fd`) writes one file per table and can dump tables in parallel with

`-j`:

```
pg_dump -d blog -Fd -j 4 -f backups/blog.dir
```

6.6 seconds, also 86 MB, with files like `4253.dat.gz` and a `toc.dat` table of contents.

Format	Flag	Output	Restore with	Pick objects	Parallel
Plain	`-Fp`	one SQL file	`psql`	no	no
Custom	`-Fc`	one compressed file	`pg_restore`	yes	restore only
Directory	`-Fd`	a folder	`pg_restore`	yes	dump and restore
Tar	`-Ft`	a tar file	`pg_restore`	yes	no

I use custom for most databases. Directory when the database is big and I want parallel dumps.

Restoring

`pg_restore -l` lists what's inside a dump:

```
245; 1259 16386 TABLE public posts blog_owner
4274; 0 0 ACL public TABLE posts blog_owner
247; 1259 16403 TABLE public tags blog_owner
...
4253; 0 16386 TABLE DATA public posts blog_owner
4255; 0 16403 TABLE DATA public tags blog_owner
```

Tables, their data, their permissions (`ACL`), then constraints and indexes. I restore into a new database, with 4 parallel jobs:

```
createdb blog_restore
pg_restore -d blog_restore -j 4 backups/blog.dump
```

6.6 seconds. Then the check that turns a hope into a backup:

```
SELECT (SELECT count(*) FROM posts) AS posts,
       (SELECT count(*) FROM post_views) AS views,
       (SELECT count(*) FROM tags) AS tags;
```

```
 posts |  views  | tags
-------+---------+------
  2193 | 5000000 |   50
(1 row)
```

Everything is there.

Restoring one table

You can restore a single table from a custom dump with `-t`. I dropped `post_tags` in the restored copy and brought it back:

```
pg_restore -d blog_restore -t post_tags backups/blog.dump
```

```
 count
-------
  2240
(1 row)
```

The rows came back. But look at the table:

```
             Table "public.post_tags"
 Column  |  Type  | Collation | Nullable | Default
---------+--------+-----------+----------+---------
 post_id | bigint |           | not null |
 tag_id  | bigint |           | not null |
```

No primary key, no foreign keys. `-t` restores the table and its data, not the constraints and indexes, which are separate items in the dump. To get everything, use `pg_restore -l` to write the list to a file, keep the lines you want, and pass it back with `-L`. It's the kind of thing you want to find out during a test, not during an emergency.

Roles are not in the dump

`pg_dump` dumps one database, and roles belong to the cluster. Restore the dump on a new server and every `GRANT` to `blog_app` fails, because `blog_app` doesn't exist there. `pg_dumpall --globals-only` dumps the roles and their settings:

```
pg_dumpall --globals-only -f backups/globals.sql
```

```
CREATE ROLE blog_app;
ALTER ROLE blog_app WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS
    ↪ CONNECTIO...
CREATE ROLE blog_owner;
ALTER ROLE blog_owner WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN NOREPLICATION
    ↪ NOBYPASSRLS;
...
ALTER ROLE blog_app SET statement_timeout TO '5s';
```

It includes the password hashes, so treat the file as a secret. Back it up next to your database dumps.

Versions must match

`pg_dump` must be at least as new as the server. My old `pg_dump` 14, the one Homebrew linked, refuses to dump the 18 server:

```
pg_dump: error: server version: 18.7 (Homebrew); pg_dump version: 14.24 (Homebrew)
pg_dump: error: aborting because of server version mismatch
```

A newer `pg_dump` can dump an older server, which is how you move data to a new major version.

Physical backups

`pg_dump` is a **logical** backup: SQL, or something close to it. A **physical** backup copies the data files of the whole cluster. `pg_basebackup` does it while the server runs:

```
pg_basebackup -D backups/base -Ft -z -P


1307035/1520221 kB (85%), 1/1 tablespace
real 42.01


-rw-------  239625     backup_manifest
-rw-------  385789963  base.tar.gz
-rw-------  414538619  pg_wal.tar.gz
```

The cluster was 1.5 GB on disk (all databases, indexes, and the write-ahead log), compressed to 800 MB in 42 seconds.

That second file, `pg_wal.tar.gz`, is the **write-ahead log**, or WAL. Every change PostgreSQL makes is first written to the WAL, then to the data files. After a crash, PostgreSQL replays the WAL to get the data files back to a consistent state. A base backup plus a continuous copy of the WAL (with `archive_command` or `pg_receivewal`) lets you restore the cluster to any moment, like 14:32 yesterday, one minute before somebody ran a `DELETE` without a `WHERE`. That's **point-in-time recovery**. Tools like pgBackRest and Barman manage it for you.

	`pg_dump`	`pg_basebackup` (+ WAL)
What	one database	the whole cluster
Size	small (no indexes, compressed)	big (everything)
Restore to	any version, any server	the same major version
Restore one table	yes	no
Point in time	no, only when it ran	yes, with WAL archiving
Restore speed for big databases	slow (rebuilds indexes)	fast

For a small application, a nightly `pg_dump` copied to another machine, plus a weekly test restore, is a solid start. Managed hosts do physical backups with point-in-time recovery for you.

Upgrading to a new major version

A minor update, like 18.6 to 18.7, only replaces the programs. The data files don't change, so you install the update and restart.

A major update, like 17 to 18, changes the format of the system catalogs, so an 18 server can't open a 17 data directory. You have three ways to move:

- `pg_dump` from the old server and restore into the new one: simple, but slow for big databases
- `pg_upgrade`: converts the cluster in place, in minutes even for big databases
- logical replication from old to new, then switch: almost no downtime, more work

I upgraded the PostgreSQL 17 cluster I used for the SQL Handbook with `pg_upgrade`.

The first try: checksums

`pg_upgrade` needs both versions installed, the old cluster stopped, and a new, empty cluster created with `initdb` from the new version. It has a `--check` mode that changes nothing. I ran it:

```
pg_upgrade -b /opt/homebrew/opt/postgresql@17/bin -B /opt/homebrew/opt/postgresql@18/bin \
  -d pg17data -D pg18up --check


Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok

old cluster does not use data checksums but the new one does
Failure, exiting
```

Remember "Data page checksums are enabled" in the `initdb` output? PostgreSQL 18 turns checksums on by default, PostgreSQL 17 didn't, and the two clusters must match. `pg_controldata` shows the setting:

```
Data page checksum version:           0
```

Two ways out: enable checksums on the old cluster with `pg_checksums --enable` (it rewrites every file, with the server stopped), or create the new cluster without them. I did the second:

```
initdb -D pg18up -U flavio --no-data-checksums
```

The `-U flavio` matters too. The 17 cluster's superuser is `flavio`, and the new cluster must have the same one. ("Checking database user is the install user" is one of the checks.)

The second try: a path too long

```
*failure*
Unix-domain socket path ".../scratch/pg/upgrade-work/.s.PGSQL.50432" is too long (maximum 103 bytes)
```

`pg_upgrade` starts both servers with their socket in the current folder, and my test folder had a very long path. Running it from `/tmp/pgup` fixed it. Not a problem you'll have on a server, but a good example of reading the log the error points to.

The upgrade

With the checks passing:

```
Checking cluster versions                                   ok
Checking database connection settings                       ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking for valid logical replication slots                ok
Checking for subscription state                             ok
Checking data type usage                                    ok
Checking for objects affected by Unicode update             ok
Checking for not-null constraint inconsistencies            ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

*Clusters are compatible*
```

I ran it for real, in `--copy` mode, which leaves the old cluster untouched:

```
Performing Upgrade
------------------
Setting locale and encoding for new cluster                 ok
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
...
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster               ok
Copying user relation files                                 ok
...
Creating script to delete old cluster                       ok
Checking for extension updates                              ok

Upgrade Complete
----------------
Some statistics are not transferred by pg_upgrade.
Once you start the new server, consider running these two commands:
    /opt/homebrew/opt/postgresql@18/bin/vacuumdb -U flavio --all --analyze-in-stages --missing-stats-
    ↪ only
    /opt/homebrew/opt/postgresql@18/bin/vacuumdb -U flavio --all --analyze-only
Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh
```

2.9 seconds. `pg_upgrade` doesn't convert your data. It recreates the schema in the new cluster and then copies (or, with `--link`, hard-links) the table files, which have the same format across versions. That's why it's fast even for big databases. With

`--link` it takes seconds whatever the size, but then the old cluster can't be used anymore once the new one starts.

The new server, with the old data:

```
 PostgreSQL 18.7 (Homebrew) on aarch64-apple-darwin27.2.0, ...

 count
-------
  2193
(1 row)
```

Statistics after the upgrade

Until PostgreSQL 17, the planner statistics were lost in an upgrade, and every query ran on guesses until you ran `ANALYZE` on everything. PostgreSQL 18 carries them over:

```
SELECT tablename, attname, n_distinct FROM pg_stats
WHERE schemaname = 'public' AND tablename = 'posts' ORDER BY 2;
```

```
 tablename |   attname    | n_distinct
-----------+--------------+-------------
 posts     | id           |          -1
 posts     | published_on | -0.79343367
 posts     | slug         |          -1
 posts     | title        |          -1
 posts     | words        |    -0.50114
(5 rows)
```

The statistics are there, from the 17 cluster. The activity counters aren't (`n_live_tup` was 0 for every table), and neither are extended statistics. That's why the output suggests `--missing-stats-only`, which only analyzes what's missing:

```
vacuumdb --all --analyze-in-stages --missing-stats-only
```

```
vacuumdb: processing database "blog": Generating minimal optimizer statistics (1 target)
...
vacuumdb: processing database "blog": Generating medium optimizer statistics (10 targets)
...
vacuumdb: processing database "blog": Generating default (full) optimizer statistics
```

An upgrade checklist

1. Read the release notes of the new version, especially the ”Migration” section.
2. Take a backup.
3. Run `pg_upgrade --check` against a copy of the cluster.
4. Do the real upgrade on the copy and run your application's tests against it.

5. Then do it for real, with the application stopped.
6. Run `vacuumdb --all --analyze-in-stages --missing-stats-only`.
7. Keep the old cluster until you're sure, then run `delete_old_cluster.sh`.

Health checks and where to host it

A database that works today can be in trouble next month: a table that grows without vacuum, a query that got slow as data grew, connections running out. These are the checks I run, all on the test server after the experiments of this book.

pg_stat_statements

The single most useful extension. It records every query the server runs, normalized (values replaced by `$1`, `$2`), with how many times it ran and how long it took. It has to be loaded at server start:

```
ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';
```

`ALTER SYSTEM` writes to `postgresql.auto.conf`, which overrides `postgresql.conf`:

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
shared_preload_libraries = 'pg_stat_statements'
idle_in_transaction_session_timeout = '30s'
log_min_duration_statement = '250ms'
```

After a restart, create the extension in the database and let the application run for a while. I ran a small workload: 200 searches, 30 dashboard queries, 5 "top posts" reports and 100 lookups by ID.

```
CREATE EXTENSION pg_stat_statements;

SELECT calls, round(total_exec_time::numeric, 1) AS total_ms,
       round(mean_exec_time::numeric, 2) AS mean_ms, rows,
       left(regexp_replace(query, '\s+', ' ', 'g'), 70) AS query
FROM pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 5;
```

```
 calls | total_ms | mean_ms | rows |                                  query
-------+----------+---------+------+-------------------------------------------------------------------------
     5 |    647.4 |  129.49 |   50 | SELECT post_id, count(*) FROM post_views GROUP BY post_id ORDER BY
    ↪  2 D
    30 |    104.5 |    3.48 |   30 | SELECT count(*) FROM post_views WHERE country = $1 AND viewed_at
    ↪ >= no
   100 |      5.0 |    0.05 |  100 | SELECT * FROM posts WHERE id = $1
   200 |      1.7 |    0.01 | 1000 | SELECT slug FROM posts WHERE search @@ websearch_to_tsquery($1, $2
    ↪ ) LI
     1 |      0.1 |    0.06 |    1 | select pg_stat_statements_reset()
(5 rows)
```

Sort by total time, not by mean. The "top posts" report ran only 5 times but took 85% of the time, because it reads all 5 million views every time. The 200 searches

cost almost nothing, thanks to the GIN index. That's where to look first: maybe that report should read from a summary table updated once an hour.

The slow query log

`log_min_duration_statement` writes every statement slower than the threshold to the server log. I set it to 100 ms and ran the report once:

```
LOG:  duration: 325.157 ms  statement: SELECT post_id, count(*) FROM post_views GROUP BY post_id ORDER
    ↪ BY 2 DESC LIMIT 3
```

A threshold between 250 ms and 1 second is a good start for a web application. Too low and the log fills with noise.

Idle in transaction

A transaction that's open and doing nothing holds its locks and stops vacuum from cleaning anything newer than its snapshot. It usually comes from code that starts a transaction and then waits for something slow, like an HTTP call, or crashes without rolling back.

`pg_stat_activity` shows them:

```
SELECT pid, state, now() - state_change AS idle_for, left(query, 30)
FROM pg_stat_activity WHERE state LIKE 'idle in transaction%';
```

```
  pid  |        state        |    idle_for     |            left
-------+---------------------+-----------------+----------------------------
 15423 | idle in transaction | 00:00:03.016748 | select count(*) from posts
(1 row)
```

The `idle_in_transaction_session_timeout` I set to 30 seconds closes such sessions automatically. Here's what the client sees when it finally tries its next query:

```
FATAL:  terminating connection due to idle-in-transaction timeout
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
connection to server was lost
```

It's brutal, but a broken connection in one client is better than a database that slowly fills with dead rows.

Cache hit ratio and database counters

`pg_stat_database` has counters for each database:

```
SELECT datname,
       round(100.0 * blks_hit / nullif(blks_hit + blks_read, 0), 2) AS hit_pct,
       xact_commit, xact_rollback, deadlocks, temp_files, pg_size_pretty(temp_bytes) AS temp
FROM pg_stat_database WHERE datname = 'blog';
```

```
 datname | hit_pct | xact_commit | xact_rollback | deadlocks | temp_files |  temp
---------+---------+-------------+---------------+-----------+------------+---------
 blog    |   97.42 |        1257 |            56 |         1 |         46 | 1491 MB
(1 row)
```

- `hit_pct`: the share of page reads served from PostgreSQL's memory. For an application database you want it above 99%. Mine is lower because I spent the afternoon scanning a 400 MB table with 128 MB of `shared_buffers`.
- `deadlocks`: 1, the one I caused on purpose.
- `temp_files`: queries that needed more memory than `work_mem` wrote 1.5 GB to temporary files. That was the index builds and the big sorts of this book. In production, temp files from normal queries mean `work_mem` is too small for them.

Tables, vacuum and wraparound

`pg_stat_user_tables` tells you, for each table, how many dead rows it has, when autovacuum last ran, and how often it's read with a sequential scan versus an index:

```
    relname     | n_live_tup | n_dead_tup |        last_autovacuum        |       last_autoanalyze
    ↪    | seq_scan | idx_scan
----------------+------------+------------+-------------------------------+------------------------------
 post_views     |    4988581 |          0 |                               | 2026-11-28
    ↪ 19:17:24.202979+01 |       75 |     2231
 post_tags      |       2240 |          0 | 2026-11-28 19:12:23.329682+01 | 2026-11-28
    ↪ 19:12:23.333706+01 |       10 |     2050
 posts          |       2193 |          0 | 2026-11-28 19:12:23.339029+01 | 2026-11-28
    ↪ 19:12:23.366324+01 |       47 |  5005053
```

A big table with many sequential scans is worth a look: it may be missing an index. PostgreSQL 18 also adds `total_vacuum_time` and `total_autovacuum_time`, so you can see how much time maintenance takes on each table.

One more check, because when it fails it's catastrophic. Transaction IDs are 32-bit numbers, and they wrap around after about 2 billion. Vacuum "freezes" old rows so this is harmless, but if vacuum can't keep up, PostgreSQL stops accepting writes to protect the data:

```
SELECT datname, age(datfrozenxid) AS xid_age,
       round(100.0 * age(datfrozenxid) / 2147483647, 4) AS pct_to_wraparound
FROM pg_database ORDER BY 2 DESC;
```

```
  datname  | xid_age | pct_to_wraparound
-----------+---------+-------------------
 postgres  |     385 |            0.0000
 template1 |     385 |            0.0000
 template0 |     385 |            0.0000
 blog      |     385 |            0.0000
(4 rows)
```

385 transactions old, so nothing to worry about on a fresh test server. Alert when it passes 50%.

Settings worth changing

The defaults are tuned for a small machine:

```
         name          | setting | unit
-----------------------+---------+------
 effective_cache_size  | 524288  | 8kB
 io_method             | worker  |
 maintenance_work_mem  | 65536   | kB
 max_wal_size          | 1024    | MB
 random_page_cost      | 4       |
 shared_buffers        | 16384   | 8kB
 work_mem              | 4096    | kB
```

`shared_buffers` is 16,384 pages of 8 kB, so 128 MB. On a dedicated server with 8 GB of RAM, these are the ones I change first:

Setting	Default	Starting point for 8 GB	Why
`shared_buffers`	128 MB	2 GB (25% of RAM)	PostgreSQL's own cache
`effective_cache_size`	4 GB	6 GB	tells the planner how much the OS caches
`work_mem`	4 MB	16 to 64 MB	memory per sort or hash, per query node
`maintenance_work_mem`	64 MB	512 MB	faster vacuum and index builds
`random_page_cost`	4	1.1	SSDs make random reads cheap
`max_wal_size`	1 GB	4 GB	fewer forced checkpoints during big writes

Be careful with `work_mem`: it's per operation, and one query can use it several times, in every connection. 100 connections with a 64 MB `work_mem` can ask for gigabytes.

Where to host it

You've seen what running PostgreSQL involves: installing, upgrading, backups, vacuum, monitoring. The question is who does it.

Managed PostgreSQL (Amazon RDS, Google Cloud SQL, DigitalOcean, Supabase, Neon, Crunchy Bridge, and many others) gives you a connection string and takes care of the server: updates, backups with point-in-time recovery, failover. It costs more than a bare server. You still own your schema, indexes, roles, queries and migrations, and everything in this book still applies.

Your own server costs less and gives you full control: any extension, any setting,

any version. You also get every task in the last four chapters, plus the ones that come with any server (security updates, disk space, a firewall). My Linux Server Handbook covers that side. And I compared the hosting options for PostgreSQL and SQLite in this post.

These questions decide it for me:

- Can I restore yesterday's backup right now, and when did I last try?
- Who notices when the server stops at 3 AM, and who fixes it?
- Does my application run as serverless functions? Then I need a pooler in front of PostgreSQL, and managed hosts usually include one.
- Do I need an extension or a setting the managed host doesn't allow?

For a side project or a first launch, I pick a managed database and spend my time on the product. When the bill grows, or I need control, I move to my own server, with `pg_dump` and `pg_restore`, the way we did in this book.