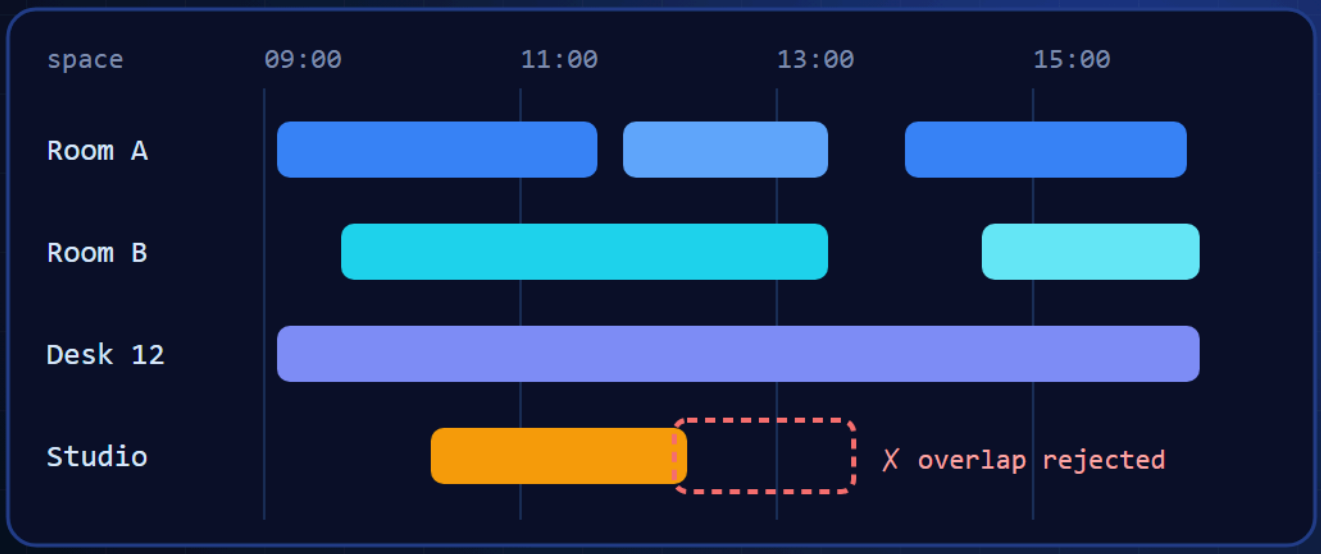



```
ALTER TABLE app.booking ADD
  EXCLUDE USING gist (space_id WITH =,
                      during WITH &&);

-- JSONB · ranges · MVCC · EXPLAIN · replication
```

A PRACTICAL GUIDE FOR DEVELOPERS

PostgreSQL Essentials

Using, Designing, and Running PostgreSQL

From psql and data types to performance, replication, and production

SQL · JSONB · INDEXES · MVCC · BACKUPS · REPLICATION

THENEXBOOKS

PostgreSQL Essentials

A Practical Guide to Using, Designing, and Running PostgreSQL
From psql and Data Types to Performance, Replication, and Production

First Edition

THENEXBOOKS

PostgreSQL Essentials
A Practical Guide to Using, Designing, and Running PostgreSQL

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. SQL and code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Test configuration changes, upgrades, and recovery procedures in a safe environment before applying them to production. PostgreSQL and its ecosystem evolve every year; always consult the official documentation for the version you run.

PostgreSQL is a registered trademark of the PostgreSQL Community Association. Other product and service names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation. This book is independent and is not affiliated with or endorsed by the PostgreSQL Global Development Group.

Nestwork, the running example in this book, is fictional. Any resemblance to real companies or services is coincidental.

Preface

PostgreSQL is a common first choice for a wide range of applications. It stores relational data with strong guarantees, but it also handles JSON documents, full-text search, geographic data, time ranges, and vector embeddings — and it can be extended in ways few other databases allow.

Using PostgreSQL *well*, however, takes more than knowing SQL. Developers who treat it as a generic database miss the features that make it powerful, such as exclusion constraints, range types, JSONB indexing, and `ON CONFLICT`. They also run into problems they do not understand: a query that is fast on a laptop but slow in production, a table that keeps growing even though rows are deleted, a migration that locks the application for minutes, or a backup that nobody has ever restored.

This book is a practical guide to PostgreSQL itself. We start with installation, `psql`, and a real schema, move through PostgreSQL's data types and advanced SQL, and then explain how PostgreSQL works inside — the planner, indexes, MVCC and vacuum, transactions and locks — so that performance stops being a mystery. Finally, we cover what it takes to run PostgreSQL in production: security, backups and point-in-time recovery, replication, partitioning, extensions, connection pooling, migrations, upgrades, and the cloud.

Who This Book Is For

- **Application developers** who use PostgreSQL through an ORM or driver and want to understand and use it properly.
- **Back-end and full-stack engineers** who design schemas, write queries, and are responsible for database performance.
- **Developers moving from another database** who want to learn what is different — and better — in PostgreSQL.
- **Developer-minded DBAs and platform engineers** who operate PostgreSQL and need a solid, practical reference.

What You Will Learn

- How to install PostgreSQL, connect securely, and work efficiently with `psql`.
- PostgreSQL's data types and constraints, including JSONB, arrays, ranges, time zones, and exclusion constraints.
- Powerful SQL: CTEs, recursive queries, `LATERAL`, window functions, upserts, `MERGE`, `COPY`, full-text search, and PL/pgSQL.
- How queries run, how to index them, and how MVCC, vacuum, transactions, and locking really work.
- Roles and row-level security, backups and point-in-time recovery, replication, partitioning, and extensions.

- Using PostgreSQL from applications, connection pooling, zero-downtime migrations, upgrades, the cloud, and a complete case study.

How This Book Is Organized

Part I — Getting Started with PostgreSQL (Chapters 1–5) installs PostgreSQL, introduces `psql`, builds the sample database, and covers SQL essentials. **Part II — Data Types and Data Modeling** (Chapters 6–10) explores PostgreSQL's type system and schema design. **Part III — Writing Powerful SQL** (Chapters 11–15) covers advanced queries, data changes, search, and server-side code. **Part IV — Performance and Concurrency** (Chapters 16–20) explains the planner, indexes, MVCC, transactions, and configuration. **Part V — Administration and Operations** (Chapters 21–25) covers security, backups, replication, partitioning, and extensions. **Part VI — PostgreSQL in Applications and Production** (Chapters 26–28) connects PostgreSQL to applications and production, ending with a full case study. The appendices provide a cheat sheet, a configuration and query reference, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Nestwork

Throughout the book we build the database for **Nestwork**, a fictional coworking company with workspaces in several cities. Members book desks and meeting rooms, hold memberships, check in with badges, and receive invoices. Room bookings, a fast-growing check-in table, and flexible member preferences give every chapter realistic data to work with.

Versions and Conventions

Examples target PostgreSQL 17 and note when a feature requires a newer or older version. Query plans and timings in examples are illustrative; your numbers will differ, and the book shows you how to measure your own. `psql` sessions show the `nestwork=>` prompt (or `nestwork=#` for a superuser).

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A mistake that can lose data or cause outages.

Contents

Preface

PART I — GETTING STARTED WITH POSTGRESQL

1 Why PostgreSQL: Features, Architecture, and Where It Fits

2 Installing PostgreSQL and Connecting: Packages, Containers, and Managed Services

3 Working Effectively with psql and GUI Tools

4 Databases, Schemas, and Tables: Building the Nestwork Database

5 SQL Essentials in PostgreSQL: Queries, Joins, and Aggregates

PART II — DATA TYPES AND DATA MODELING

6 PostgreSQL Data Types: Numbers, Text, Dates and Times, Booleans, UUIDs, Enums, and Domains

7 Constraints and Data Integrity: Keys, CHECK, UNIQUE, Exclusion Constraints, and Generated Columns

8 JSONB, Arrays, and Composite Types

9 Range Types, Time Zones, and Scheduling Data

10 Data Modeling for PostgreSQL: Normalization, Naming, and Schema Design Patterns

PART III — WRITING POWERFUL SQL

11 Advanced Queries: CTEs, Recursive Queries, LATERAL, and Subqueries

12 Window Functions and Analytics

13 Changing Data: RETURNING, ON CONFLICT, MERGE, and Bulk Loading with COPY

14 Full-Text Search and Fuzzy Matching

15 Functions, Procedures, Triggers, and PL/pgSQL

PART IV — PERFORMANCE AND CONCURRENCY

16 How PostgreSQL Executes Queries: The Planner, EXPLAIN, and EXPLAIN ANALYZE

17 Indexing in Depth: B-tree, GIN, GiST, BRIN, Partial, and Expression Indexes

18 MVCC, VACUUM, and Autovacuum

19 Transactions, Isolation Levels, and Locking

20 Configuration, Monitoring, and pg_stat_statements

PART V — ADMINISTRATION AND OPERATIONS

21 Roles, Privileges, and Row-Level Security

22 Backup and Recovery: pg_dump, pg_basebackup, WAL Archiving, and Point-in-Time Recovery

23 Replication and High Availability: Streaming and Logical Replication

24 Partitioning and Managing Large Tables

25 Extensions and the Ecosystem: pgcrypto, pg_trgm, PostGIS, pgvector, and Foreign Data Wrappers

PART VI — POSTGRESQL IN APPLICATIONS AND PRODUCTION

26 PostgreSQL from Application Code: Drivers, Connection Pooling, Migrations, and ORMs

27 Security Hardening, Upgrades, and Running PostgreSQL in the Cloud

28 Case Study: Building and Operating the Nestwork Database End to End

APPENDICES

A psql and SQL Cheat Sheet

B Configuration and Query Reference

C Production Checklist

D Glossary

PART I

Getting Started with PostgreSQL

What PostgreSQL is and how it is built, how to install it and connect securely, how to work efficiently in psql, and the SQL essentials — all on a real sample database.

1 · Why PostgreSQL
2 · Installing and Connecting
3 · Working Effectively with psql
4 · Building the Nestwork Database
5 · SQL Essentials in PostgreSQL

CHAPTER 1 · PART I — GETTING STARTED WITH POSTGRESQL

Why PostgreSQL: Features, Architecture, and Where It Fits

Before we install anything, we look at what PostgreSQL actually is: a community-owned, extensible relational database with a long history and a very particular internal design. Understanding that design — processes, shared memory, the write-ahead log, and the files on disk — will make every later chapter easier to follow.

What PostgreSQL Is

PostgreSQL is an open source, object-relational database management system. That one sentence packs in three ideas that shape everything about how the system behaves and how you will use it.

Open source and community-owned. PostgreSQL is released under the PostgreSQL License, a short, permissive license in the same family as the BSD and MIT licenses. You can use it, modify it, embed it in a commercial product, and ship it to customers without paying anyone or publishing your changes. Just as important, no single company owns the project. Development is coordinated by the PostgreSQL Global Development Group, a community of individual contributors and companies that employ them. Several vendors sell support, hosting, or enhanced distributions, but none of them controls the core code base or its roadmap. For a database — something you may run for a decade or more — that independence is a practical property, not just a philosophical one: the project cannot be bought, relicensed, or discontinued by a single business decision.

Relational and standards-leaning. At its heart PostgreSQL is a relational database that speaks SQL. The project takes the SQL standard seriously: when the standard defines a feature, PostgreSQL usually implements the standard syntax rather than inventing its own, and the documentation marks where behavior deviates. Identity columns, `MERGE`, `JSON_TABLE`, window functions, and recursive common table expressions all follow standard syntax. Where the standard is silent, PostgreSQL adds its own extensions — `RETURNING`, `ON CONFLICT`, `DISTINCT ON`, and many operators — and is clear about which is which.

Extensible by design. The "object" part of object-relational is about extensibility. PostgreSQL stores the definitions of data types, operators, functions, index access methods, and procedural languages in its own system catalogs, and new ones can be added at run time without changing the server's source code. This is why an extension can add a geographic type with spatial indexes, a vector type for similarity search, or a foreign data wrapper that queries another system — and why those additions feel native rather than bolted on. Extensions are the subject of Chapter 25, but you will see their influence throughout the book.

Transactional to the core

PostgreSQL is fully transactional. Every statement runs inside a transaction, and transactions have the familiar ACID properties: atomicity, consistency, isolation, and durability. Less obviously, almost everything in PostgreSQL is transactional, including most DDL. You can create a table, add an index, alter a column, and then roll the whole thing back:

```
nestwork=# BEGIN;
BEGIN
nestwork=*# CREATE TABLE app.scratch (id int PRIMARY KEY, note text);
CREATE TABLE
nestwork=*# ALTER TABLE app.scratch ADD COLUMN created_at timestamptz DEFAULT now();
ALTER TABLE
nestwork=*# ROLLBACK;
ROLLBACK
nestwork=# SELECT to_regclass('app.scratch');
 to_regclass
-------------

(1 row)
```

The table never existed as far as any other session is concerned. Transactional DDL makes schema migrations far safer: a migration script can run inside a single transaction and either apply completely or not at all. A few commands, such as `CREATE DATABASE`, `VACUUM`, and `CREATE INDEX CONCURRENTLY`, cannot run inside a transaction block, and we point them out when they appear.

Concurrency is handled by **multiversion concurrency control** (MVCC). Instead of locking a row while someone reads it, PostgreSQL keeps multiple versions of rows and shows each transaction a consistent snapshot. Readers do not block writers, and writers do not block readers. The price of this design is that old row versions accumulate and must be cleaned up by `VACUUM`. Chapter 18 is devoted to that trade-off, and Chapter 19 covers isolation levels and locking.

A feature tour

The short version, with pointers to where each topic is covered:

Area	Highlights	Chapter
Data types	numeric, text, timestamptz, UUID, enums, domains	6
Integrity	CHECK, UNIQUE, foreign keys, exclusion constraints	7
Semi-structured	JSONB, arrays, composite types	8
Ranges	tstzrange, daterange, multiranges	9
Query power	CTEs, LATERAL, window functions	11, 12
Writes	RETURNING, ON CONFLICT, MERGE, COPY	13
Search	Full-text search, trigram matching	14

Server code	PL/pgSQL functions, procedures, triggers	15
Indexing	B-tree, GIN, GiST, BRIN, partial, expression	17
Security	Roles, privileges, row-level security	21
Operations	PITR, streaming and logical replication	22, 23
Scale	Declarative partitioning	24

What ties these together is that features compose: a range type works with GiST indexes, which power exclusion constraints, which are enforced transactionally under MVCC. Much of the craft of using PostgreSQL well is knowing how to combine them.

A Brief History

PostgreSQL's roots are academic. It began as the POSTGRES research project at the University of California, Berkeley, in the mid-1980s, led by Michael Stonebraker as a successor to his earlier Ingres work — the name literally means "post-Ingres". The research system explored ideas that were unusual at the time: user-defined types, rules, and a storage model that kept old versions of data around rather than overwriting it in place. Echoes of all three survive in today's PostgreSQL.

POSTGRES originally used its own query language, PostQUEL. In the mid-1990s, two graduate students replaced it with an SQL interpreter and released the result as Postgres95. Shortly afterwards the project was renamed PostgreSQL to reflect its SQL support, and development moved from the university to an open, distributed community. The name is pronounced "post-gress-Q-L", and "Postgres" remains an accepted short name — you will see both in documentation, package names, and conversation.

Since then the project has shipped a major release roughly every year, adding point-in-time recovery, streaming replication, JSONB, logical replication, declarative partitioning, and parallel query along the way — always conservative about correctness and cautious about breaking compatibility.

> **NOTE**
>
> The executable is called `postgres`, the default superuser is usually called `postgres`, and the default maintenance database is also called `postgres`. Context tells you which one is meant. In this book "PostgreSQL" refers to the product, and `postgres` in code font refers to the program, role, or database.

Versions and Release Cadence

Knowing how PostgreSQL versions work saves you from confusing upgrades and from running unsupported software without realizing it.

Major and minor versions

Since PostgreSQL 10, a version number has two parts. The first number is the **major version** (10, 11, … 17, 18) and the second is the **minor version** (17.0, 17.1, 17.2, …). Before version 10, the major version was the first *two* numbers, so 9.5 and 9.6 were different major versions and 9.6.3 was a minor release of 9.6. You will still meet the old scheme in older documentation and long-lived systems.

The distinction matters because the two kinds of release are very different:

- **Major releases** arrive about once a year, traditionally in the autumn. They add features and may change the on-disk format of the system catalogs. Moving between major versions requires an upgrade procedure — `pg_upgrade`, a dump and restore, or logical replication — which Chapter 27 covers.
- **Minor releases** contain only bug fixes and security fixes. They never change the data format, so applying one is a matter of installing new binaries and restarting the server. The project publishes minor releases on a regular schedule, several times a year, and out of band when an urgent security issue requires it.

Each major version is supported with minor releases for five years after its initial release. After that it reaches end of life and receives no more fixes, including security fixes. A reasonable operational policy is to apply minor releases promptly and plan major upgrades well before the end-of-life date.

> **TIP**
> Minor releases are low-risk and high-value. Treat them like operating system security updates: schedule them, read the release notes for any post-upgrade steps (occasionally an index needs to be rebuilt), and apply them. Running an old minor release of a supported major version gives you all of the known bugs and none of the fixes.

Which version this book uses

The examples in this book target **PostgreSQL 17**. When a feature needs a particular version, we say so — for example, `MERGE` arrived in PostgreSQL 15 and `JSON_TABLE` in PostgreSQL 17. PostgreSQL 18, released in 2025, adds features such as the `uuidv7()` function, virtual generated columns, and asynchronous I/O; we mention these briefly where they are relevant and label them "PostgreSQL 18". You can check the version you are connected to at any time:

```
nestwork=# SHOW server_version;
 server_version
----------------
 17.6
(1 row)

nestwork=# SELECT current_setting('server_version_num')::int AS version_num;
 version_num
```

```
-------------
      170006
(1 row)
```

Your minor version will differ. The numeric form, `server_version_num`, is convenient for scripts that need to compare versions: 170006 means major 17, minor 6.

Where PostgreSQL Fits

No database is the right answer to every problem, and it is worth being clear about where PostgreSQL is a strong fit and where another tool may serve you better. The comparisons below are about design choices, not about which product is "best".

Compared with other relational databases

MySQL and MariaDB are the other widely used open source relational databases. They are mature, well understood, and power a great many web applications. Historically they emphasized simplicity and read-heavy web workloads, while PostgreSQL emphasized SQL completeness, strictness about data validity, and extensibility. The gap on many individual features has narrowed over the years, so the choice today often depends on team experience, ecosystem, and the specific features you need — rich types, exclusion constraints, transactional DDL, or a particular extension may tip the balance toward PostgreSQL.

Commercial databases such as Oracle Database and Microsoft SQL Server offer broad feature sets and vendor support. PostgreSQL covers much of the same ground for typical application workloads, but migrations are rarely mechanical: procedural languages, locking behavior, and optimizer hints differ. **SQLite** is an embedded library rather than a server — ideal for a mobile app or a single-process tool — so the two are complementary more than competitive.

Compared with non-relational systems

Document stores, search engines, time-series databases, and analytical column stores each optimize for a particular access pattern. PostgreSQL overlaps with several of them — JSONB, full-text search, and extensions cover many such needs — but overlap is not replacement. A useful rule of thumb: start with PostgreSQL as the system of record, and add a specialized system when you can name a concrete, measured requirement that PostgreSQL does not meet well, such as frequent ad hoc scans of billions of rows that suit a columnar engine.

What PostgreSQL asks of you

PostgreSQL's strengths come with responsibilities. Its connection model — one operating-system process per connection — means that thousands of mostly idle connections are expensive, so busy applications need a connection pooler (Chapter 26). MVCC means dead row versions must be

vacuumed, so you need to understand autovacuum (Chapter 18). And the default configuration is deliberately conservative so that PostgreSQL starts on almost any machine, which means a production server needs some tuning (Chapter 20). None of this is difficult, but none of it is optional either.

Architecture Overview

With the "why" covered, let's look at the "how". The rest of this chapter describes what is running on a PostgreSQL server, what lives in memory, and what is written to disk. Every one of these pieces reappears later in the book.

Clusters and databases

PostgreSQL uses the word **cluster** in a way that surprises newcomers. A *database cluster* is not a group of servers; it is a single PostgreSQL server instance: one data directory, managed by one running server process tree, listening on one port. A cluster contains one or more **databases**. Some objects belong to the cluster as a whole — roles, tablespaces, and the list of databases itself — while everything else, such as schemas, tables, functions, and extensions, belongs to exactly one database.

```
Database cluster  (data directory /var/lib/postgresql/17/main, port 5432)
|
+-- Cluster-wide objects: roles, tablespaces, database list
|
+-- database "postgres"    (default maintenance database)
+-- database "template1"   (template for new databases)
+-- database "template0"   (pristine template, never modified)
+-- database "nestwork"
      +-- schema "app"      (city, location, space, member, booking, ...)
      +-- schema "billing"  (invoice, invoice_line)
      +-- schema "public"
```

A client connection is always to one database. You cannot write a query that joins a table in `nestwork` with a table in another database of the same cluster, at least not without an extension such as a foreign data wrapper. This is why PostgreSQL applications usually use **schemas**, not databases, to organize related objects: schemas are namespaces within a database, and queries can freely reference several of them. Chapter 4 creates the `nestwork` database and its `app` schema.

The three databases that exist in every new cluster have specific roles. `template1` is copied whenever you run `CREATE DATABASE` without naming a template, so anything you add to it appears in every new database. `template0` is a pristine copy that you should never modify; you use it when you need a new database with a different encoding or locale. `postgres` is simply a convenient default database for tools and administrators to connect to.

The process model

PostgreSQL uses a multi-process architecture rather than a multi-threaded one. When the server starts, a single supervisor process starts first. Historically it was called the **postmaster**, and the name survives in the documentation, in the `postmaster.pid` file, and in everyday conversation, although the binary is `postgres`. The postmaster allocates shared memory, starts a set of auxiliary processes, and then listens for client connections.

Each time a client connects, the postmaster forks a new **backend process** dedicated to that connection. The backend authenticates the client, then parses, plans, and executes every query that client sends, until the client disconnects and the process exits. Backends never serve more than one connection, and connections never move between backends.



This design has consequences. If one backend crashes, the postmaster terminates the others and reinitializes shared memory to be safe. And each connection costs a whole process, which is why a pooler becomes essential when connection counts grow. The default `max_connections` is 100; raising it far beyond what the hardware can run concurrently usually makes things worse.

You can see the processes from the operating system. On a Debian or Ubuntu server the process titles include the cluster name:

```
$ ps -o pid,cmd -u postgres
    PID CMD
   1021 /usr/lib/postgresql/17/bin/postgres -D /var/lib/postgresql/17/main -c conf...
   1024 postgres: 17/main: checkpointer
   1025 postgres: 17/main: background writer
   1027 postgres: 17/main: walwriter
   1028 postgres: 17/main: autovacuum launcher
```

```
   1029 postgres: 17/main: logical replication launcher
   4410 postgres: 17/main: nestwork_app nestwork 10.20.0.15(51544) idle
   4415 postgres: 17/main: nestwork_app nestwork 10.20.0.15(51560) SELECT
```

Backend titles show the role, database, client address, and state. The same information, and much more, is available inside the database through `pg_stat_activity`:

```
nestwork=# SELECT pid, backend_type, usename, datname, state
nestwork-#   FROM pg_stat_activity
nestwork-#  ORDER BY backend_type, pid;
 pid  |         backend_type         |   usename    | datname  | state
------+------------------------------+--------------+----------+--------
 1028 | autovacuum launcher          |              |          |
 1025 | background writer            |              |          |
 1024 | checkpointer                 |              |          |
 4410 | client backend               | nestwork_app | nestwork | idle
 4415 | client backend               | nestwork_app | nestwork | active
 4502 | client backend               | postgres     | nestwork | active
 1029 | logical replication launcher | postgres     |          |
 1027 | walwriter                    |              |          |
(8 rows)
```

The auxiliary processes

The auxiliary processes do work that should not be done by client backends, either because it must happen continuously or because it would add latency to user queries:

- **Checkpointer** — periodically performs a *checkpoint*: it writes all modified ("dirty") pages from shared buffers to the data files and records a point in the WAL from which crash recovery can start. Checkpoints are triggered by time (`checkpoint_timeout`, five minutes by default) or by the amount of WAL generated (`max_wal_size`).
- **Background writer** — trickles dirty pages out to disk between checkpoints so that backends needing a free buffer rarely have to write one themselves.
- **WAL writer** — flushes the write-ahead log buffers to disk at short intervals, which matters mostly for transactions that use asynchronous commit.
- **Autovacuum launcher** — decides which tables need vacuuming or analyzing and starts **autovacuum workers** to do it. Chapter 18 explains why this is essential.
- **Logical replication launcher** — starts workers for logical replication subscriptions, if any exist (Chapter 23).
- **WAL senders and WAL receivers** — stream WAL to and from replicas when replication is configured.
- **Archiver** — copies completed WAL segment files to an archive when WAL archiving is enabled, the basis of point-in-time recovery (Chapter 22).
- **Parallel workers** and other **background workers** — started on demand for parallel query or by extensions that register their own background processes.

PostgreSQL 17 also added a **WAL summarizer** process, which runs when `summarize_wal` is enabled and records which blocks changed so that `pg_basebackup --incremental` can take incremental backups. Older versions had a separate statistics collector process; since PostgreSQL 15, cumulative statistics are kept in shared memory instead, so you will not see that process on a current server.

Shared memory and shared buffers

All backends and auxiliary processes attach to a common region of shared memory. The largest part of it is **shared buffers**, PostgreSQL's own cache of data pages. PostgreSQL stores tables and indexes in fixed-size pages of 8kB, and every read or write of table data goes through shared buffers: to read a row, a backend finds the page in shared buffers or reads it from the operating system into a free buffer; to modify a row, it changes the page in shared buffers and marks it dirty. Dirty pages are written back to the data files later by the background writer, the checkpointer, or occasionally a backend.

The size is set with the `shared_buffers` parameter, which defaults to 128MB in stock builds — deliberately small so the server starts anywhere. PostgreSQL also relies on the operating system's page cache, so a page can be cached twice: once in shared buffers and once in the kernel. This "double buffering" is why the usual guidance is to give shared buffers a meaningful fraction of RAM on a dedicated server, but not the majority of it. Chapter 20 discusses how to choose a value and how to measure whether it is working, using views such as `pg_stat_io` (PostgreSQL 16 and later) and `pg_statio_user_tables`.

Shared memory also holds the WAL buffers, the lock table, and the list of running transactions that MVCC needs to build snapshots. Each backend additionally has private memory for sorts and hashes, governed by `work_mem`, which is allocated per operation — another reason connection counts and memory settings must be considered together.

The write-ahead log

If modified pages are written to the data files lazily, what protects a committed transaction if the server loses power a second after `COMMIT` returns? The answer is the **write-ahead log** (WAL), and it is the single most important idea in PostgreSQL's storage design.

The rule is simple: before a change to a data page may be written to disk, a record describing that change must already be safely written to the WAL. When a transaction commits, PostgreSQL writes a commit record to the WAL and, by default, waits until the WAL has been flushed to durable storage before reporting success. The data pages themselves can be written much later. If the server crashes, the startup process replays the WAL from the last checkpoint forward, reapplying every change, and the database returns to a consistent state that includes every committed transaction.

```
UPDATE app.space SET hourly_rate = 38.00 WHERE space_id = 3;  COMMIT;

1. backend modifies page in shared buffers (page is now "dirty")
2. backend writes a WAL record describing the change into WAL buffers
```

```
  3. at COMMIT, WAL up to the commit record is flushed to pg_wal/  (fsync)
  4. client receives COMMIT                 <-- durable from this point
  5. later: background writer / checkpointer writes the dirty page
  6. after a checkpoint, older WAL is no longer needed for crash recovery
```

Sequential appends to a log are cheaper than scattered writes to many data files, so WAL is efficient as well as safe. The same stream of WAL is what makes replication and point-in-time recovery possible. A replica is, in essence, a server that is permanently in recovery, replaying WAL shipped from the primary (Chapter 23). An archive of WAL segments plus a base backup lets you replay history up to any moment you choose (Chapter 22). Logical decoding reads the WAL to produce a stream of row changes for logical replication and change data capture.

WAL is stored in the `pg_wal` subdirectory as a series of segment files, each 16MB by default. Every position in the WAL is identified by a **log sequence number** (LSN), which you will see in replication and backup views:

```
nestwork=# SELECT pg_current_wal_lsn(), pg_walfile_name(pg_current_wal_lsn());
 pg_current_wal_lsn |     pg_walfile_name
--------------------+--------------------------
 0/3A6F1E28         | 000000010000000000000003A
(1 row)
```

> **WARNING**
>
> Never delete files from `pg_wal` by hand to free disk space, and never turn off `fsync` on a server whose data you care about. Both break the promise that makes crash recovery work, and the result can be a cluster that will not start or, worse, one that starts with silently corrupted data. If `pg_wal` is growing, find out why — usually a failing archive command, an inactive replication slot, or a very large `max_wal_size` — and fix the cause.

The Data Directory

Everything a cluster stores lives under a single **data directory**, often referred to as `PGDATA` after the environment variable that tools use to find it. Its location depends on how PostgreSQL was installed — Chapter 2 lists the usual locations — but you can always ask the server:

```
nestwork=# SHOW data_directory;
       data_directory
-----------------------------
 /var/lib/postgresql/17/main
(1 row)
```

Viewing the data directory requires superuser privileges or membership in the `pg_read_all_settings` role, because it reveals details about the server's file system. The top level of a data directory looks like this:

```
$ sudo -u postgres ls /var/lib/postgresql/17/main
PG_VERSION    pg_dynshmem   pg_notify     pg_snapshots  pg_subtrans   pg_wal
base          pg_logical    pg_replslot   pg_stat       pg_tblspc     pg_xact
global        pg_multixact  pg_serial     pg_stat_tmp   pg_twophase   postgresql.auto.conf
pg_commit_ts                                                          postmaster.opts
                                                                      postmaster.pid
```

The entries you should recognize:

Entry	Contents
`PG_VERSION`	Major version that created the cluster
`base/`	One subdirectory per database
`global/`	Cluster-wide catalogs (roles, databases)
`pg_wal/`	Write-ahead log segment files
`pg_xact/`	Commit status of transactions
`pg_tblspc/`	Links to tablespace directories
`pg_replslot/`	Replication slot state
`pg_stat/`	Statistics saved at clean shutdown
`postgresql.auto.conf`	Settings written by ALTER SYSTEM
`postmaster.pid`	PID, port, and socket of a running server

On Red Hat-style systems and in many source builds, the main configuration files `postgresql.conf`, `pg_hba.conf`, and `pg_ident.conf` are also in the data directory. Debian and Ubuntu packages keep them under `/etc/postgresql/17/main/` instead. Chapter 2 shows how to find them on any system.

Before PostgreSQL 10, `pg_wal` was called `pg_xlog` and `pg_xact` was `pg_clog`; they were renamed because people mistook them for disposable log files.

How a table maps to files

Inside `base/`, each database has a directory named after its object identifier (OID), and each table or index is stored in files named after its *relfilenode* number. You rarely need to look at these files, but knowing that they exist demystifies a lot. You can ask PostgreSQL where a table lives:

```
nestwork=# SELECT pg_relation_filepath('app.booking');
 pg_relation_filepath
----------------------
 base/16384/16502
(1 row)

nestwork=# SELECT pg_size_pretty(pg_relation_size('app.booking'))       AS heap,
nestwork-#        pg_size_pretty(pg_total_relation_size('app.booking')) AS total;
```

```
  heap   |  total
---------+---------
 1024 kB | 1648 kB
(1 row)
```

The numbers will be different on your system. A table's main file is split into 1GB segments (`16502`, `16502.1`, ...). Alongside the main data file you will find a **free space map** (`16502_fsm`), which records where there is room for new rows, and a **visibility map** (`16502_vm`), which records pages whose rows are all visible to every transaction — information that lets `VACUUM` skip pages and lets index-only scans avoid visiting the table. The main table storage is called the **heap**, because rows are stored in no particular order. Rows larger than about 2kB have their big values compressed and, if necessary, moved to a separate TOAST table, which is why a table with large text or JSONB values may have a total size much larger than its heap size.

> **NOTE**
>
> Relfilenode numbers change when a table is rewritten — by `TRUNCATE`, `VACUUM FULL`, `CLUSTER`, or some forms of `ALTER TABLE`. Never hard-code them in scripts; always look them up with `pg_relation_filepath()`. And never copy individual table files between clusters: without the matching catalog entries, transaction status, and WAL they are meaningless.

What Happens When You Run a Query

Putting the pieces together, here is the life of a single query from the Nestwork application: "show the meeting rooms at the Alfama location".

1. The driver sends the SQL text to the connection's backend process.
2. The **parser** checks syntax; the **analyzer** resolves names and types using the system catalogs.
3. The **rewriter** expands views and applies row-level security policies.
4. The **planner** estimates the cost of alternative plans using statistics gathered by `ANALYZE` and picks the cheapest.
5. The **executor** runs the plan, reading pages through shared buffers and checking each row version against the transaction's snapshot.

```
nestwork=# EXPLAIN SELECT s.name, s.capacity
nestwork-#   FROM app.space AS s
nestwork-#   JOIN app.location AS l USING (location_id)
nestwork-#  WHERE l.name = 'Nestwork Alfama' AND s.kind = 'meeting_room';
                                QUERY PLAN
---------------------------------------------------------------------------
 Hash Join  (cost=1.05..2.23 rows=1 width=36)
   Hash Cond: (s.location_id = l.location_id)
   ->  Seq Scan on space s  (cost=0.00..1.14 rows=3 width=40)
         Filter: (kind = 'meeting_room'::text)
   ->  Hash  (cost=1.04..1.04 rows=1 width=4)
```