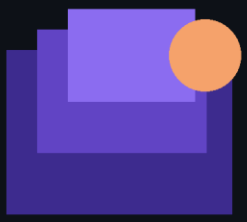

A DEVELOPER'S GUIDE TO
POSTGRESQL 18 IN PRODUCTION

# The PostgreSQL Handbook

From First Query to Production

```
-- Semantic search: your relational database, one extension away
CREATE INDEX kb_article_embedding_hnsw
    ON kb_article USING hnsw (embedding vector_cosine_ops);

SELECT title, body,
       embedding <=> $1 AS distance
FROM kb_article
WHERE metadata @> '{"verified": true}'
ORDER BY embedding <=> $1
LIMIT 5;

-- RETURNING makes every write a query
INSERT INTO ticket (customer_id, subject, priority)
VALUES ($1, $2, 'urgent')
RETURNING id, reference, created_at;
```

**RACHID DAHIR**

POSTGRESQL 18

# The PostgreSQL Handbook

*From First Query to Production*

First Edition · 2026

**Rachid Dahir**

**AviSoft**

Technical Publishing

---



First Edition · 2026



avisoft.ma

# Preface

## Who This Book Is For

This book is written for developers who already talk to a database and want to understand the one on the other end of the connection string. You can write a SELECT with a WHERE clause and a JOIN, your application works, and yet the database itself stays a black box: you are not sure why one query is fast and its near-twin is slow, what a transaction actually guarantees under concurrent writers, or which of PostgreSQL's capabilities you are paying for and never using. If that describes you — backend, full-stack, or somewhere in between — this book was written for you.

No prior PostgreSQL expertise is assumed. Comfort with basic SQL is enough to start; everything past SELECT, WHERE, and JOIN is built on the page, in order, against one running example. .NET developers get a specific welcome: the last three chapters speak C# directly, connecting the SQL fluency the book builds to Npgsql, EF Core, and pgvector. But this is not a C# book. The first twenty chapters contain no C# at all, and a reader who never opens Visual Studio loses nothing before Chapter 21. If your SQL habits were formed on MySQL or SQL Server, you will feel at home quickly; the places where PostgreSQL behaves differently are called out as they arrive.

You may also be the accidental keeper of a production database: the developer who inherited “the Postgres box” along with the on-call rotation. Parts IV and V were written with your pager in mind — indexes, query plans, security, backups, partitioning, and replication are the chapters that turn inherited responsibility into informed operation.

## What This Book Covers

Twenty-three chapters in five parts, followed by five appendices. Every part works the same running example, a support-ticketing system called Lumina Helpdesk, so nothing here is a toy schema invented for one page and abandoned on the next. The five parts move from getting a server running, through relational fluency, into the capabilities that are the reason to choose this database, and out the far side into performance, operations, and application code. The Introduction walks the parts chapter by chapter.

## Get the Most Out of This Book

This is a book meant to be read with a keyboard within reach. Chapter 1 gets a server running in minutes, and from then on the fastest way through any chapter is to run what you read. Four production decisions make that worth your time:

- Every SQL listing is real, executable SQL, verified against a live PostgreSQL 18.4 database by an automated harness before it was allowed into print. The output printed under a listing was captured from an actual run, never hand-typed.
- The database is deterministic by design: every seed script uses a fixed random seed and a fixed timestamp anchor, so when you run a listing you get the book's printed output exactly — not "output may vary," but the same rows in the same order on your machine.
- One script, `scripts/reset-to-chapter.ps1 -Chapter N` on Windows or `scripts/reset-to-chapter.sh N` on macOS and Linux, rebuilds the database to the exact state chapter N expects. Start reading anywhere, experiment freely, and reset when an experiment goes sideways — a clean, correct starting point is always one command away. The Introduction describes what else the companion repository ships.

## Where to Get the Code

Everything the book runs against lives in one public repository: `github.com/RachidD68/the-postgresql-handbook`. It ships the Lumina Helpdesk schema and migrations, the deterministic seed data, every SQL listing in the book as a runnable file, all 69 exercise solutions, the reset and verification scripts, and the small C# solution behind Chapters 21 to 23. The code is MIT-licensed and free to use in commercial work — licensed separately from the text of this book.

## Conventions Used

- `inline code` refers to a table, column, function, keyword, or small SQL fragment.
- Block SQL uses Cascadia Code with light-IDE-style syntax colors, so a query on the page reads like a query in your editor. Keywords are uppercase throughout, following the style guide in Appendix C — the same contract every listing in the book obeys.

- Each listing carries a numbered caption, and for SQL listings the same number names its .sql file in the companion repository — finding the runnable copy is a filename lookup. Shell, configuration, and C# listings are printed to be read; the C# ones live under src/.
- Output blocks reproduce `psql`'s real formatting — aligned columns, row counts and all — because they are captured from `psql`, not typeset by hand.
- 💡 **Pro Tip**, ⚠️ **Warning**, 🔍 **Going Deeper**, 🌍 **Real-World Scenario**, and 📝 **Key Takeaway** callouts mark content that earns a break from the narrative; 🔗 **Cross-Reference** points to related ground elsewhere in the book, and 🧪 **Try It** invites you to the keyboard.
- Two callouts are specific to this book: ⚖️ **Standard SQL vs PostgreSQL** flags where PostgreSQL extends or diverges from the SQL standard, and 🕒 **Version Note (16 → 18)** flags behavior that changed since PostgreSQL 16, collected for reference in Appendix E.

# Introduction

## Why This Book Exists

PostgreSQL is the most capable open-source database ever shipped, and most of what is written about it treats it as a place to practice generic SQL. The tutorials teach SELECT, JOIN, and GROUP BY as if the engine underneath were interchangeable — and for those three keywords, it nearly is. But nobody chooses PostgreSQL because it runs standard SQL. People choose it for what the standard does not cover: JSONB documents inside relational rows, window functions that answer analytical questions in one pass, an MVCC engine that lets readers and writers ignore each other, declarative partitioning, row-level security enforced by the database rather than the application, and now vector search through pgvector. Those are precisely the topics most SQL material skips, because they do not transfer to the next database over.

This book teaches PostgreSQL as PostgreSQL, not as a dialect. Where the standard and PostgreSQL agree, it says so; where PostgreSQL goes further, that is usually the point of the chapter. And every claim is grounded in one running system: Lumina Helpdesk, a support-ticketing database that Chapter 2 designs and every later chapter extends, queries, indexes, secures, partitions, and finally searches semantically. When Chapter 16 shows you a query plan, it is a plan over data you have; when Chapter 19 partitions a table, it is a table you watched grow past half a million rows.

## How This Book Is Structured

| Part | Chapters | Focus |
|---|---|---|
| Part I — Foundations | Ch 1–5 | Installation, the Lumina schema, CRUD, result shaping, and data types. |
| Part II — Relational Thinking | Ch 6–10 | Constraints, joins, aggregation, subqueries and CTEs, window functions. |

| | | |
|---|---|---|
| Part III — The PostgreSQL Difference | Ch 11–14 | JSONB, search, server-side code, and MVCC concurrency. |
| Part IV — Performance & Operations | Ch 15–20 | Indexes, EXPLAIN, security, backups, partitioning, replication. |
| Part V — PostgreSQL in the Real World | Ch 21–23 | Application code, plus gateway chapters to EF Core and pgvector. |

## Part I — Foundations (Chapters 1–5)

Chapter 1 answers why PostgreSQL is worth a whole book and gets a server running on your machine, Docker-first, with your first psql conversation at the end of it. Chapter 2 designs Lumina Helpdesk — tickets, customers, agents, teams, comments, tags, and SLA policies — and that schema is the ground every later chapter builds on. Chapter 3 covers the four verbs of daily work: inserting, reading, updating, and deleting. Chapter 4 shapes results with filtering, sorting, pagination, and expressions. Chapter 5 closes the part with data types done right, because the cheapest moment to choose the correct type is before the data arrives.

## Part II — Relational Thinking (Chapters 6–10)

Chapter 6 puts the database on defense: constraints that make bad data impossible to insert rather than expensive to clean up. Chapter 7 reads across tables with joins, building each kind from a question that needs it. Chapter 8 turns rows into answers with aggregation and grouping. Chapter 9 composes larger queries from smaller ones with subqueries and CTEs. Chapter 10 finishes with window functions — rankings, running totals, and comparisons across rows that would otherwise require exporting data to a spreadsheet.

## Part III — The PostgreSQL Difference (Chapters 11–14)

This part is the book's namesake argument: four chapters of capability that separate PostgreSQL from a generic relational database. Chapter 11 stores JSONB documents inside relational rows and indexes them, so the relational-versus-document choice stops being either-or. Chapter 12 builds search, from LIKE patterns through trigram

matching to full-text search with ranking. Chapter 13 moves logic into the server itself: views, functions, procedures, and triggers, with honest guidance on when each earns its keep. Chapter 14 explains MVCC without tears — what actually happens when two transactions touch the same row, and why PostgreSQL's answer is the reason the previous thirteen chapters worked so calmly.

### Part IV — Performance & Operations (Chapters 15–20)

By this part, Lumina's deterministic bulk seed has grown the database to 80,040 tickets and 521,561 events, so the performance chapters measure something real instead of toy tables. Chapter 15 matches index types to the shape of each question. Chapter 16 reads the planner's mind with EXPLAIN and turns plans into tuning decisions. Chapter 17 covers security: roles, privileges, and row-level security that holds even against a connected user. Chapter 18 handles backups, maintenance, and monitoring — the chapter you want read before you need it. Chapter 19 partitions the event table when one table is no longer enough, and Chapter 20 adds replication and connection pooling, the two production staples between your application and everything else you have built.

### Part V — PostgreSQL in the Real World (Chapters 21–23)

Chapter 21 crosses from psql to application code — connections, parameters, pooling from the client side, and the injection mistakes that parameterized queries exist to end. Chapters 22 and 23 are deliberately labeled gateways: they are bridges to two other books, not tutorials on those tools. Chapter 22 shows what the SQL you now read fluently looks like when EF Core generates it, then hands off to The Entity Framework Core Handbook for that tool's full story. Chapter 23 adds pgvector and closes the book with its flagship: hybrid search that fuses Chapter 12's full-text ranking with vector similarity through reciprocal rank fusion, the payoff of a promise made seven chapters earlier. Where that road continues into production retrieval systems belongs to Production-Grade RAG with C# and .NET, and the chapter says so plainly.

## The Appendices

- **Appendix A — Installing PostgreSQL 18 Natively:** the no-Docker path, from installer questions to a verified first connection.

- **Appendix B — The psql Survival Guide:** the client's metacommands and habits, organized for the moment you need them.
- **Appendix C — SQL Style Guide:** the binding style contract every listing in this book follows.
- **Appendix D — Worked Solutions:** all 69 practice exercises solved and explained, each verified by the same harness as the chapters.
- **Appendix E — What Changed: PostgreSQL 16 → 18:** a version-delta reference for readers arriving from an older server.

## The Companion Project

**The companion repository:** `github.com/RachidD68/the-postgresql-handbook`

Clone it once and the whole book is runnable. The schema and seed data live under schema/ and seed/, every SQL listing is a standalone file under sql/ at the exact path the book prints, the worked exercise solutions are under exercises/, and tools/ contains the verification harness itself. The only C# in the repository sits under src/ — three small projects that serve Chapters 21 through 23 and nothing else.

Lumina Helpdesk grows across the book the way real systems grow. Chapter 2 designs the core schema, and from that point it is frozen: every later change ships as an additive migration, never a rewrite, so nothing you learned goes stale. Chapter 15 loads the bulk seed that takes the database to 80,040 tickets and 521,561 events. Chapter 23 adds a knowledge base with embedded articles, ready for vector search. At every step, the state is deterministic — seeded with a fixed random seed and a fixed timestamp anchor, so your database is not merely similar to the book's; it is identical to it.

One script holds it all together: `scripts/reset-to-chapter.ps1 -Chapter N` on Windows, or `scripts/reset-to-chapter.sh N` on macOS and Linux. It rebuilds the database to the exact state chapter N expects to find. It makes the book safe to read out of order and safe to experiment against, because no mistake survives a reset.

> **Going Deeper**
>
> *Chapter 1's primary path is Docker, and the companion compose file uses the pgvector/pgvector:pg18 image for every chapter — deliberately, so a reader who starts with*

*it in Chapter 1 never hits a wall in Chapter 23, which needs the vector extension that image carries. Appendix A is the native-install alternative for readers who prefer PostgreSQL as a Windows service; native readers reach Chapter 23 with the same compose file, since pgvector does not ship with the native installer.*

## A Note on This Edition

This first edition targets PostgreSQL 18.4. If you are on 16 or 17, nearly everything still applies, and Appendix E enumerates exactly what changed between 16 and 18 so you know which pages to read with a footnote in mind.

One production claim deserves to be stated plainly: every SQL listing in this book was executed against a live PostgreSQL 18.4 database by an automated harness, and every printed output was captured from a real run. The listings that are meant to fail were verified to fail with exactly the error the prose predicts; only a handful of environment-level commands, and the shell and C# listings, are printed to be read rather than run. Combined with the deterministic seeds, that means the output on the page is not approximate, illustrative, or lightly edited — it is what the database said. The discipline was not free, and it paid for itself: more than once during production, a captured output refused to match a claim in the prose, and the prose was wrong. Those corrections are in the book you are holding instead of in an errata list.

**Key Takeaway**

*You are reading a book whose every listing has already run, in order, on a database identical to yours. Nothing here is pseudocode, nothing is simplified past the point of running, and nothing depends on state the book did not build in front of you.*

## Let's Begin

Chapter 1 starts at the beginning: what a relational database actually is, why this one won, and your first conversation with a running server. If you have PostgreSQL experience already, it will read quickly and hand you a working environment; if you do not, it is the calmest on-ramp in the book. Either way, by its final page you will have a

server answering your queries, and the remaining twenty-two chapters will never let it sit idle.

Turn the page.

# Contents

A map of the book — by Part, then Chapter, with each chapter's sections and page numbers listed beneath it.

## Part I — Foundations

# Back Matter

## About the Author

# Rachid Dahir

*Software Architect • Technical Author • Educator*

Rachid Dahir designs systems for the long haul. A software architect, technical author, and educator based in Morocco, he has spent over a decade building enterprise-grade solutions in financial services, healthcare, and large-scale enterprise environments — where reliability, performance, and clean architecture are not optional.

His books are written for developers who are ready to move beyond tutorials: rigorous, production-grounded, and clear enough that advanced topics never feel out of reach. Whether you are a mid-career developer leveling up, a database administrator consolidating best practices, or a team standardizing on modern PostgreSQL, his goal is the same: clarity without compromise.

### Teaching Philosophy

Rachid's approach to technical education rests on three principles:

- **Production-grade code only.** Every sample is written as if it belongs in a real system — no toy examples, no shortcuts that would never survive a code review.
- **Explain, demonstrate, practice.** Durable understanding comes from doing, not just reading. Concepts are shown in context, then reinforced through hands-on exercises.
- **No gatekeeping — only clear teaching.** Advanced topics deserve the same patience and rigor as introductory ones. The complexity of a subject is never an excuse for a poor explanation.

When he is not writing code or prose, Rachid explores mathematics, AI research, and natural health.

### Connect with Rachid


- **GitHub** RachidD68 (Rachid DAHIR)
- **Email** rachiddahir@avisoft.ma

# Part I

## Foundations

*Why PostgreSQL, getting it running with Docker or a native install, and designing the Lumina Helpdesk schema this whole book builds on. By the end of Part I you can insert, read, update, and delete data confidently, and you know why every column's type was chosen on purpose.*

# Chapter 1 — Why PostgreSQL, and Getting It Running

*What a relational database actually is, why this one won, and your first conversation with it.*

> **Real-World Scenario**
>
> *PostgreSQL runs more of your day than you suspect. It has been the database behind Instagram's early growth and Skype's backend, it sits under countless SaaS products whose names you know, and it has topped Stack Overflow's developer survey as the most admired and most wanted database for years running. None of that came from marketing — PostgreSQL has no company, no sales team, and no license fee. It got there by being the database that engineers, given a free choice, keep choosing.*

This book teaches PostgreSQL from the first query to a production-shaped database, through one running example you will come to know like a colleague: the helpdesk of a SaaS company called Lumina. This chapter has two jobs. The first is orientation: what a relational database actually is and why this particular one earned its reputation. The second is practical: by the last page you will have PostgreSQL 18 running on your machine, and you will have spoken to it.

## The filing cabinet with a librarian

Imagine a filing cabinet. Folders in drawers, drawers in the cabinet: it stores things, and if you are disciplined, you can find them again. A surprising amount of software is written as if a database were exactly this — a place where bytes go to be safe. But a filing cabinet answers no questions. It does not know that two folders describe the same customer, it will happily hold contradictory copies of a phone number, and when someone asks "which invoices from March are unpaid?", the cabinet's contribution is to be openable.

A relational database is the cabinet plus a librarian — a fast, obsessive, slightly pedantic librarian. You describe your data's shape once: these are customers, these are tickets, every ticket belongs to exactly one customer. From then on the librarian enforces the shape, answers questions of arbitrary complexity, handles a thousand people reading and writing at once without letting them trample each other, and survives a power cut mid-sentence without losing a word. The language you share with this librarian is SQL, and the librarian's half of the conversation is a pipeline worth seeing once now, because the whole book lives inside it:

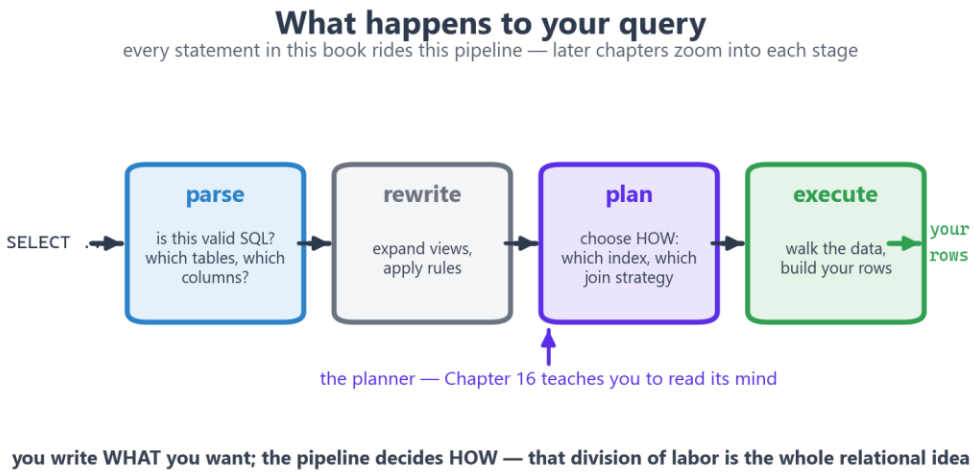


*Figure 1.1 — Parse, rewrite, plan, execute. You declare what you want; the stages decide how.*

The pipeline embodies the relational idea's one radical commitment: you state what you want, never how to get it. You will not write loops that walk files; you will describe a result, and the planner — the pipeline's third stage and, by Chapter 16, an old acquaintance — chooses the strategy. That division of labor is why the same query keeps working as your data grows a thousandfold: the what stays fixed while the database quietly upgrades the how.

## Where PostgreSQL came from, and why it won

PostgreSQL is old enough to have an origin story with chalkboards in it. It began in 1986 as POSTGRES, Michael Stonebraker's research project at Berkeley — the successor to his earlier Ingres system, hence the name. It picked up SQL and its current

name in the mid-1990s, went fully open source under a permissive license, and has shipped a major release roughly every year since under the care of a global volunteer core team. No company owns it. Version 18, this book's target, arrived in September 2025 — the fortieth-ish anniversary of a grad-school project that outlived nearly every commercial rival it was compared against.

Longevity alone is not a recommendation; plenty of old software survives on inertia. PostgreSQL won on three repeated bets. It bet on correctness — standards fidelity, honest transactions, data that does not silently truncate or coerce — back when faster-but-looser rivals were winning benchmarks. It bet on extensibility: types, functions, indexes, even whole query behaviors can be added by extensions, which is why when the AI era suddenly needed vector search, PostgreSQL grew it in months (Chapter 23 is the payoff). And it bet on zero friction: no license fees, no editions, no sales call. The whole database, from your laptop to a petabyte warehouse, is the same free artifact, and ecosystems compound around bets like those.

## An honest comparison

A book that tells you its subject is always the right choice is a brochure. Here is the fair version of the neighborhood:

| Database | Where it genuinely shines | The honest fine print |
|---|---|---|
| MySQL / MariaDB | Read-heavy web apps; enormous hosting ecosystem | Historic quirks in strictness; replication story matured earlier than PG's |
| SQL Server | Deep Microsoft-stack shops; superb tooling | Licensing cost shapes architecture; Windows-centric gravity |
| SQLite | Embedded, mobile, single-user tools — brilliant at it | One writer at a time; a library, not a server; not built for concurrent apps |
| PostgreSQL | Nearly everything else, and increasingly the default | Connection model wants a pooler at scale (Ch. 20); no vendor to shout at |

So when is PostgreSQL the wrong choice? When the database lives inside a phone app or a CLI tool, SQLite wins and it is not close. When your team is fluent in SQL Server and the licenses are already paid, momentum is a real engineering asset. And when your problem is truly a cache or a queue of ephemeral blobs, a relational database of any brand may be ceremony you do not need — though Chapter 14 will show a job queue living happily inside PostgreSQL anyway. For the wide middle — applications with structured data, real questions, and more than one user — the rest of this book is the argument.

> **Standard SQL vs PostgreSQL**
>
> *"Standard SQL" is a spectrum, not a place. Every database implements the core of the ISO SQL standard plus its own dialect: extra types, extra clauses, different escape hatches. PostgreSQL sits unusually close to the standard and documents its departures; this book flags notable ones in boxes exactly like this, so what you learn travels with you.*

## Setting up: one command, whole environment

Everything in this book runs against PostgreSQL 18 in a Docker container, defined by one file in the companion repository. Docker gives every reader of this book — Windows, macOS, Linux, Apple Silicon or not — the identical database, which means when your output matches mine it is because we are truly running the same software. If you prefer a native installation, Appendix A walks through it, and every chapter works the same way. Here is the heart of the file:

```
# CodeSample/docker-compose.yml (abridged)
services:
  db:
    image: pgvector/pgvector:pg18
    container_name: lumina-db
    environment:
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-lumina}
    ports:
      - "5432:5432"
    volumes:
      - lumina-data:/var/lib/postgresql/data

  pgadmin:
    image: dpage/pgadmin4:latest
```

```
    ports:
      - "8080:80"
```

*Listing 1.1 — PostgreSQL 18 plus pgAdmin. The full file adds health checks, tuned settings, and env-var overrides for ports and passwords.*

One line deserves its footnote now rather than in Chapter 23: the image is `pgvector/pgvector:pg18` — built on the official PostgreSQL 18 image, with one extension pre-installed. You will not touch that extension for twenty-two chapters, but because it is here from day one, the database you set up this morning is the same one that does semantic search in the finale. No mid-book reinstall, no wall.

Clone the repository, stand in it, and bring the environment up:

```
git clone https://github.com/RachidD68/the-postgresql-handbook.git
cd the-postgresql-handbook
docker compose up -d
```

```
[+] Running 4/4
 ✓ Network the-postgresql-handbook_default   Created
 ✓ Volume the-postgresql-handbook_lumina-data  Created
 ✓ Container lumina-db       Started
 ✓ Container lumina-pgadmin  Started
```

*Listing 1.2 — The -d runs it in the background. First run downloads the images; later runs take seconds.*

Two services are now living on your machine. The database listens on port 5432, the port PostgreSQL has answered on since before some of its users were born. And pgAdmin, a browser-based management UI, waits at http://localhost:8080 (log in with admin@lumina.local / lumina, then register a server pointing at host db, user postgres, password lumina). pgAdmin is genuinely useful for browsing and the occasional visual sanity check, and this book will now say almost nothing more about it — because the terminal client is about to earn your loyalty.

## First contact with psql

psql is PostgreSQL's command-line client: the fastest route from a question to an answer, the tool every tutorial and Stack Overflow answer assumes, and the interface this book speaks throughout. The container ships it, so you need install nothing:

```
docker exec -it lumina-db psql -U postgres
```

*Listing 1.3 — Run psql inside the container, as the superuser postgres. Your prompt becomes postgres=#.*

With that command you became the second kind of client in Figure 1.2 — pgAdmin was the first, and Chapter 21's application code will be the third.

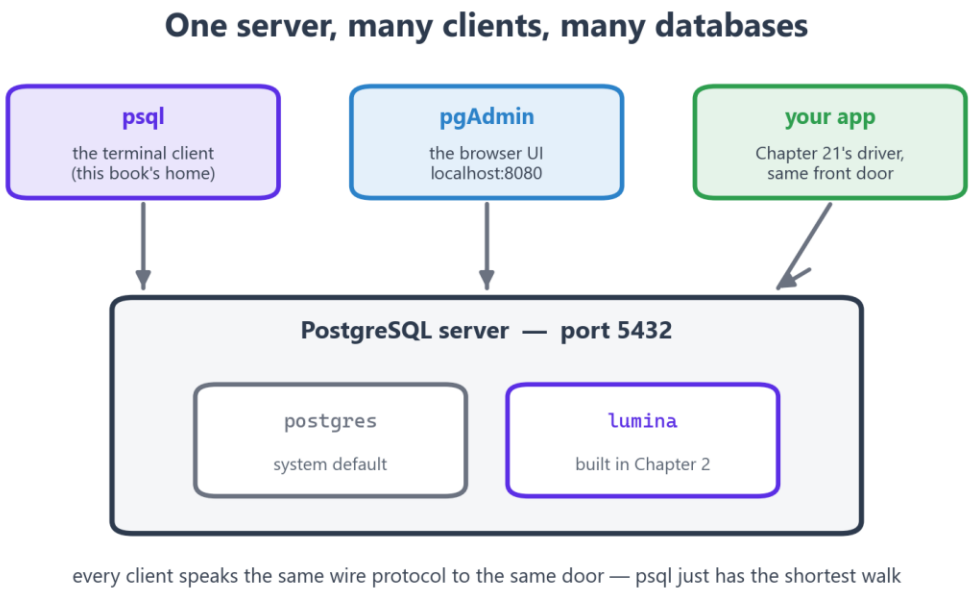


*Figure 1.2 — psql, pgAdmin, and eventually your application all knock on the same door: port 5432.*

That prompt — postgres=# — is three facts in one glance: you are connected to the database named postgres (a system default that every fresh server carries), = means psql awaits a new statement, and # confesses you are a superuser, which is fine on a laptop and a Chapter 17 conversation anywhere else. Ask the server to introduce itself:

```
SELECT version();
```

```
                                  version
-------------------------------------------------------------------------------
 PostgreSQL 18.4 (Debian 18.4-1.pgdg13+1) on x86_64-pc-linux-gnu, compiled by gcc ...
(1 row)
```

*Listing 1.4 — Your exact line varies with chip and image build — what matters is the 18.*

Note the semicolon: psql sends nothing until it sees one, which is how multi-line statements stay comfortable. And notice that a SELECT needs no table at all — the projection Chapter 3 will formalize is already at your service, calculator included:

```
SELECT 1 + 1 AS obligatory, 6 * 7 AS better;
```

```
 obligatory | better
------------+--------
          2 |     42
(1 row)
```

*Listing 1.6 — Header, row, row count: the anatomy of every result grid you will ever read.*

## The meta-commands you'll use daily

Anything starting with a backslash is not SQL — it is an instruction to psql itself, and a handful of them will be in your fingers within a week. The first is `\l`, list databases:

```
\l
```

```
                                List of databases
   Name    |  Owner   | Encoding | ... | Access privileges
-----------+----------+----------+-----+-------------------
 postgres  | postgres | UTF8     | ... |
 template0 | postgres | UTF8     | ... |
 template1 | postgres | UTF8     | ... |
(3 rows)
```

*Listing 1.8 — A fresh server: the default database and two templates it clones new ones from.*

Three databases and none of them ours — which is the cue to introduce the companion repository's most important script. Lumina's database gets designed in Chapter 2 and seeded in Chapter 3, but you do not have to replay history by hand every time you sit down. From the repository root, in a regular terminal (not inside psql):

```
.\scripts\reset-to-chapter.ps1 -Chapter 1

# macOS / Linux readers:  ./scripts/reset-to-chapter.sh 1
```

*Listing 1.10 — Reset the database to exactly the state a chapter expects. -Chapter 1 is a clean, nearly-empty start.*

That script is the book's time machine. Every chapter opens from a known state: run it with the chapter's number and the database contains precisely what the chapter assumes — the schema as of that point, the seed data, nothing stale. Break something while experimenting (please do), and the reset makes it never have happened. It is also why every printed data result in this book reproduces on your machine byte for byte; the few environment-shaped outputs, like the version banner earlier, say so in their

captions. The script finds psql on your machine if you have it, and otherwise runs it inside the container for you, so it works either way. Reconnect and look around your new, nearly empty home:

```
\c lumina
\dt

You are now connected to database "lumina" as user "postgres".
Did not find any tables.
```

*Listing 1.11 — Connected to lumina; no tables yet. Chapter 2 fixes that personally.*

The rest of the daily kit, briefly: `\d thing` describes a table (its columns, types, and constraints — you will run this constantly from Chapter 2 on), `\q` quits, and `\?` lists every meta-command when memory fails. Appendix B is the full tour, including output formatting and scripting tricks.

> **Pro Tip**
>
> *Learn \x before you need it. It toggles expanded display — each column on its own line — which turns a wide row from an unreadable wraparound into a tidy card. The first time a table with fifteen columns hits your terminal, \x is the difference between reading and squinting.*

## A tour of the companion repository

You cloned it for the compose file; here is what else you now own. The repository is the book's second half — every listing, runnable; every dataset, loadable; every chapter, resettable:

| Path | What lives there |
|---|---|
| docker-compose.yml | This chapter's environment: PostgreSQL 18 + pgAdmin |
| scripts/ | reset-to-chapter and friends — the time machine |
| schema/ | Lumina's tables (Ch. 2) plus one migration per schema-changing chapter |

| | |
|---|---|
| seed/ | The deterministic cast: 40 tickets whose stories fill the whole book |
| sql/ch01 … ch23/ | Every numbered listing, exactly as printed |
| exercises/ | Solutions to each chapter's exercises (hints stay in-chapter) |
| tools/ | The verification harness the book was built with |
| src/ | The small C# solution for Chapters 21–23 — SQL-first until then |
| config/ | The settings the book's captured output was produced under |

One habit to adopt now: when a chapter's output differs from yours, reset to that chapter before suspecting the book or your sanity. Nearly every discrepancy is state — an experiment left over, a chapter replayed twice — and the reset erases the question. The listings in sql/ are generated from the book's own source and verified against a live server before printing; if something still disagrees after a reset, the repository's issue tracker wants to hear about it.

> **Cross-Reference**
>
> *Prefer PostgreSQL installed directly on your machine? Appendix A covers native installs on Windows, macOS, and Linux, and every script honors the standard PGHOST / PGUSER environment variables, so nothing else in the book changes. The full psql handbook — formatting, scripting, the meta-command catalog — is Appendix B.*

> **Key Takeaway**
>
> *A relational database is not storage with extra steps — it is a system that enforces your data's shape and answers declarative questions about it. You now have the best open-source implementation of that idea running locally, a client to converse with it, and a repository that can reconstruct any chapter's world on demand.*

## Summary

You started with the filing-cabinet-and-librarian picture of what a relational database adds to mere storage: enforced shape, declarative questions, safe concurrency, durability. You traced PostgreSQL from a Berkeley research project to the most admired database in the industry, saw the three bets — correctness, extensibility, zero friction — that got it there, and gave its rivals their honest due, including the cases where SQLite or an existing SQL Server investment wins.

Then you built the book's world: one docker compose up for PostgreSQL 18 and pgAdmin, first contact through psql, the anatomy of a prompt and a result grid, and the daily meta-commands. The companion repository's reset script is the piece to remember — every chapter opens from a known state, every printed result is reproducible, and no experiment can break anything for longer than one command takes to run.

## Key Terms

- **Relational database** — storage plus enforcement: data with a declared shape and a query engine over it.
- **SQL** — the declarative language for describing what you want; the pipeline decides how.
- **Server / client** — PostgreSQL is a long-running server process; psql, pgAdmin, and your app are clients.
- **Cluster** — one running PostgreSQL server and the databases it hosts.
- **psql** — the terminal client, and this book's home.
- **Meta-command** — a backslash instruction to psql itself (\l, \d, \x), not SQL.
- **Container** — the isolated, identical-everywhere environment the compose file defines.
- **Planner** — the pipeline stage that chooses how to answer a query; Chapter 16's subject.

# Practice Exercises

## Exercise 1.1 — Basic: Poke the server

Connect with psql, list the databases, connect to lumina, and ask the server three questions: its version, the current time (SELECT now();), and the name of the database you are connected to (SELECT current_database();). Read each result grid aloud: header, rows, count.

*Hint:* every answer is one meta-command or one single-line SELECT from this chapter.

## Exercise 1.2 — Intermediate: Read a system catalog

PostgreSQL describes itself in tables. Run \d pg_database, pick two columns whose purpose you can guess from their names, then select them — the shape you need is SELECT datname FROM pg_database; where FROM names the table to read (Chapter 3 gives FROM its proper introduction; consider this a preview). You have just queried the librarian's own card catalog — the same skill Chapter 18 uses to monitor production databases.

*Hint:* swap datname for any column \d showed you. Wide output? This is what \x was born for.

## Exercise 1.3 — Challenge: Break it and come back

Stop the database out from under yourself: docker compose stop, then try a query and read the error psql gives you — learning what "the database is down" actually looks like is best done when it is fake. Bring it back with docker compose start, reconnect, and confirm with one query that your data survived. Then, for the full disaster drill: docker compose down -v (the -v deletes the data volume), bring it up again, and use the reset script to rebuild the world.

*Hint:* stop/start never touches your data; the named volume is why even a full docker compose down (without -v) keeps it. The -v flag is why backups exist; Chapter 18 takes that thread seriously.

## What's Next

You have a running server and an empty prompt — what you lack is a database worth querying. Chapter 2 fixes that the honest way: starting from Lumina's actual problem, a support inbox drowning in a shared spreadsheet, and designing the ten tables that will carry every example to the end of the book. You will meet the team, the customers, and the first of many opinions about naming things. Bring the container; the napkin sketch is waiting.

# Chapter 2 — Designing Lumina Helpdesk: Databases, Schemas, and Tables

*From a napkin sketch to CREATE TABLE: the database this whole book lives in.*

> **Real-World Scenario**
>
> *Lumina is a SaaS company with a problem you may recognize. Support requests arrive by email, chat, web form, and the occasional desperate phone call. They land in a shared inbox, get copied into a spreadsheet named helpdesk_FINAL_v3.xlsx, and are assigned to agents by whoever notices first. Last month a customer asked why nobody answered her urgent ticket. The answer, which no one wanted to say out loud, was that two people had each assumed the other had it. This chapter is where we fix that — not with an app, but with the database an app deserves.*

## Lumina has a problem

Every example in this book runs against one database: the helpdesk we are about to design for Lumina. You will meet the same people again and again. Priya Nair leads Technical Support and has opinions about API rate limits. Sofia Ramos runs Billing and has seen every way an invoice can go wrong. Amara Diallo handles Onboarding, which means she answers the same three questions forty times a week and does it kindly every time. On the other side sit customers like Fiona O'Brien of Harbor Analytics and Jack Chen of Northwind Traders, who file tickets, wait for answers, and occasionally rate the experience one star out of five.

Why a helpdesk? Because tickets are a gift of a teaching domain. They connect people to conversations, conversations thread into trees, workloads beg to be counted and ranked, and every ticket drags a small cloud of loosely structured facts behind it. By the

end of the book this one schema will have carried joins, recursive queries, full-text search, JSON documents, job queues, and even semantic search. Today it just needs to exist.

Right now, Lumina's "database" is a spreadsheet. Before we design the replacement, it is worth being precise about why the spreadsheet fails. Not because spreadsheets are beneath us — they run more of the world's finance than anyone admits — but because the specific ways this one fails will tell us exactly what the fix looks like.

## Why not one big spreadsheet

Here is a slice of helpdesk_FINAL_v3.xlsx, rebuilt as a single database table. One row per ticket, and every fact about that ticket crammed into the same row: who filed it, their email, which agent picked it up, which team that agent belongs to. It looks harmless. Type it in and watch closely.

```
-- Lumina's spreadsheet, faithfully reproduced as one wide table.
-- (Even the INSERT is sloppy — no column list. It fits the exhibit.)
CREATE TABLE helpdesk_sheet (
    ticket_ref     text,
    subject        text,
    customer_name  text,
    customer_email text,
    agent_name     text,
    agent_team     text
);

INSERT INTO helpdesk_sheet VALUES
    ('LUM-1003', 'Invoice shows wrong VAT number',
     'Fiona O''Brien', 'fiona@harboranalytics.example', 'Sofia Ramos', 'Billing'),
    ('LUM-1015', 'Updating card details fails with 402',
     'Fiona O''Brien', 'fiona@harboranalytics.example', 'Sofia Ramos', 'Billing'),
    ('LUM-1027', 'Printer-friendly view cuts off table',
     'Fiona O''Brien', 'fiona@harboranalytics.example', 'Marcus Webb', 'Technical
Support');
```

*Listing 2.1 — The one-big-table design. It works right up until it doesn't.*

Fiona's name appears three times. Her email appears three times. Sofia's team affiliation appears twice. None of that is data — it is the same three facts, photocopied. And photocopies drift. When Fiona changes her email address, you must find and update every row she has ever appeared on. Miss one, and the database now holds two contradictory claims about her email, with no way to tell which is true. Database people

call this an update anomaly, and it has cousins: delete Fiona's last ticket and you have deleted Fiona, since nowhere else records that she exists.

There is a subtler smell too. What is this table about? Tickets? Customers? Agents? It cannot decide. A table that answers three questions at once answers none of them well. The fix is old, boring, and effective: split the sheet so that each table is about exactly one kind of thing, and each fact lives in exactly one place. A customer table owns Fiona's email. An agent table owns Sofia's team. A ticket table owns the connection between them, and nothing else.

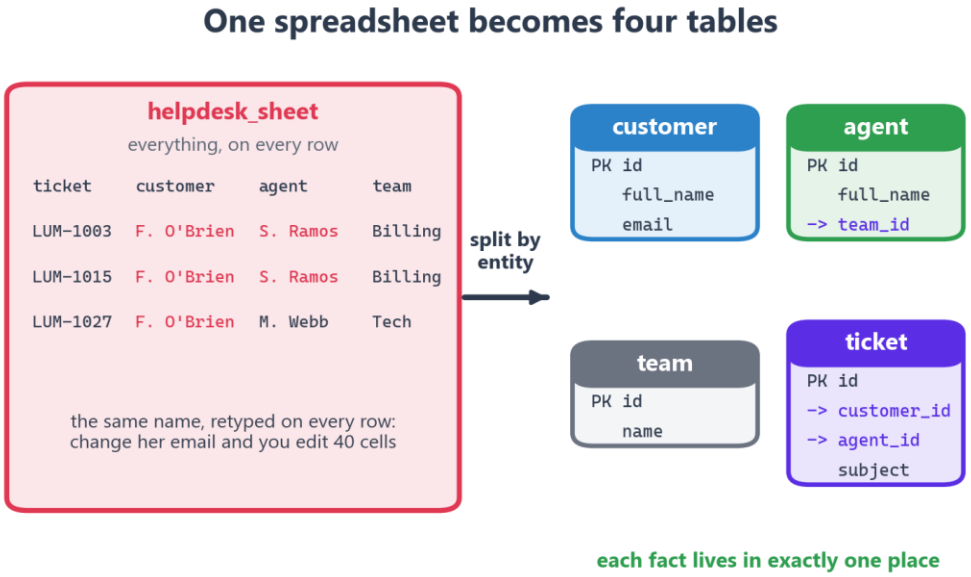


*Figure 2.1 — The wide sheet splits by entity. Repetition disappears because each fact now has one home.*

Textbooks teach this as normalization, complete with numbered normal forms and formal definitions of functional dependency. You can learn those later if the mood strikes; I have shipped a lot of schemas and reach for the theory about once a decade. The working rule that covers most of it: if you find yourself typing the same fact twice, you are probably missing a table. Train that reflex and the normal forms mostly take care of themselves.

> **Key Takeaway**
>
> *One table per kind of thing, one home per fact. Repetition in a schema is not a style problem — it is a standing invitation for the data to contradict itself.*

## Databases, schemas, and tables

Before we build the real tables, a quick map of where tables actually live, because PostgreSQL has one more layer than most people expect. A running PostgreSQL server is called a cluster, and it can hold many databases, each fully isolated from the others. Inside a database, tables are grouped into schemas — named folders, nothing more mysterious than that. So the full address of a table reads like a file path: cluster, then database, then schema, then table.

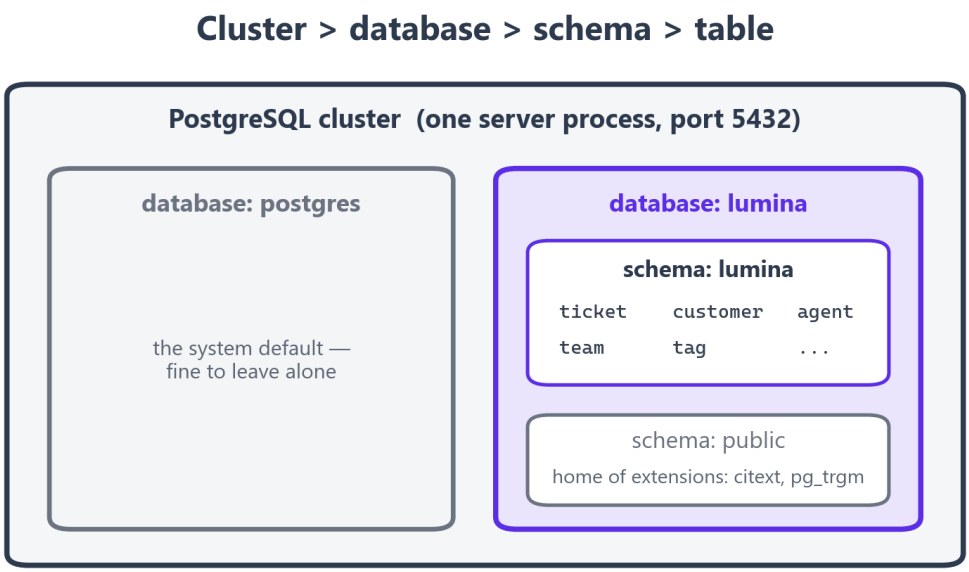


*Figure 2.2 — One server, many databases; one database, many schemas. Our tables live in the lumina schema of the lumina database.*

Your container came with a database named postgres. It is the system default, useful as a place to connect when nothing else exists yet, and we will leave it alone. Lumina gets its own database:

```
CREATE DATABASE lumina;
```

*Listing 2.2 — One database for the whole book. You only run this once.*

Reconnect with `\c lumina` and everything you create from now on lands there. One honesty note: if you ran the reset script in Chapter 1, lumina already exists — that is how you connected to it there — and running CREATE DATABASE again just earns you a harmless "already exists" complaint. The statement is here because you should know the spell, and because a reader on a bare server needs it for real.

Now the schema. Every database is born with one called public, and plenty of projects dump everything into it and live happily. We are going to be slightly more deliberate and give Lumina's tables a schema of their own. It costs one line now and pays for itself the first time a second application, an extension, or a migration tool wants to share the database without tripping over our tables.

```
CREATE SCHEMA lumina;
```

*Listing 2.3 — A named home for every Lumina table.*

How does PostgreSQL decide which schema an unqualified table name refers to? It walks a setting called search_path, in order, and uses the first match. Check yours, then put lumina at the front. (Reset-script users will find lumina already first; the note after the listing explains why.)

```
SHOW search_path;

SET search_path TO lumina, public;
```

*Listing 2.4 — With lumina first in the path, plain table names resolve to our schema.*

In this chapter I will write the schema out in full — `lumina.ticket`, every time — because you are learning what the dots mean. From Chapter 3 onward the book writes bare table names and lets search_path do its job. The companion database pins the setting permanently with `ALTER DATABASE lumina SET search_path`, so it survives reconnects.

### Standard SQL vs PostgreSQL

*The word SCHEMA is the least portable word in databases. In PostgreSQL and the SQL standard it means a namespace inside a database. In MySQL, SCHEMA is literally a synonym for DATABASE. In Oracle, a schema is a user's personal namespace. When someone says "schema" in a mixed-database meeting, ask which one they mean — half the room is picturing something different.*

## From entities to tables

Back to the napkin. Circle the nouns in Lumina's problem and you get the cast of the schema: teams, agents, customers, tickets, the comments people exchange on a ticket, the tags that categorize it. Each noun becomes a table. The art is in choosing the columns, and three conventions will carry us the whole way.

- **Singular table names, snake_case everywhere.** A table is a set of things, but each row is one thing: one agent, one ticket. Write ticket, not tickets, and never SmartCase — PostgreSQL folds unquoted names to lowercase anyway, and fighting that leads to a life of double quotes.
- **id, then <table>_id.** Every table gets a numeric primary key named id. A column that points at another table borrows its name: customer_id points at customer, team_id at team. You can read a schema's whole shape from its column names alone.
- **Timestamps end in _at, dates end in _on.** created_at is an instant; hired_on is a calendar day. The suffix tells you the type before you look.

> ⚠ **Warning**
>
> *Naming is the one decision in this chapter you will not get to revisit cheaply. Columns and tables get baked into queries, application code, reports, and other people's muscle memory. Renaming a table two years in is a project, not a statement. Spend the extra minute now.*

That line about snake_case and lowercase deserves proof, because the alternative looks so reasonable at first. Suppose a well-meaning colleague, fresh from another database, creates a table with the casing they are used to:

```
-- Looks fine. Is a trap.
CREATE TABLE "TicketNote" (
    "NoteId"  bigint,
```

```
    "Body"    text
);

-- Now every query must quote, forever, with exactly this casing:
SELECT "NoteId", "Body" FROM "TicketNote";

-- While the unquoted name never worked at all — it folds to lowercase:
SELECT NoteId FROM TicketNote;
```

*Listing 2.5 — The quoted table exists and answers; its unquoted name does not exist at all.*

Here is the rule underneath the trap. PostgreSQL folds every unquoted identifier to lowercase before looking it up, so TicketNote, ticketnote, and TICKETNOTE all mean ticketnote. Double quotes switch that folding off and make the name case-sensitive, permanently. The table "TicketNote" is sitting right there — the quoted SELECT just read from it — yet the final SELECT fails with a relation-not-found error, because it asked for ticketnote, which was never created. Every tool, every ORM, and every colleague who forgets the quotes fails the same way. I have watched teams carry that tax for years because the rename never quite made it to the top of the backlog. Lowercase snake_case names cost nothing and never need quoting. That is the entire argument, and it is sufficient. All that remains is disposing of the evidence, which needs the quotes one last time:

```
DROP TABLE "TicketNote";
```

*Listing 2.6 — Even the cleanup is case-sensitive. This is no way to live.*

Conventions settled, here are the first two tables. The support organization: teams, and the agents who belong to them.

```
CREATE TABLE lumina.team (
    id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name        text NOT NULL,
    focus_area  text NOT NULL,
    created_at  timestamptz NOT NULL
);

CREATE TABLE lumina.agent (
    id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    team_id     bigint NOT NULL,
    manager_id  bigint,
    full_name   text NOT NULL,
```

```
    email       citext NOT NULL UNIQUE,
    hired_on    date NOT NULL
);
```

*Listing 2.7 — Teams and agents. manager_id points back into agent itself.*

Several choices here deserve a sentence each. GENERATED ALWAYS AS IDENTITY tells PostgreSQL to hand out the id numbers itself, counting up, and to refuse your help; you will never invent an agent id by hand, and that is a feature. NOT NULL marks the columns that must have a value for the row to make sense — an agent without a name is not an agent. And manager_id has no NOT NULL, which is our first deliberate use of emptiness: team leads have no manager, and for them the column simply holds nothing. Chapter 4 will show you how much careful thinking that little word NULL demands.

The odd one out is `citext` — case-insensitive text, from an extension the Chapter 1 reset script installed for you (the manual spell is CREATE EXTENSION citext). Emails are the classic case: `Priya.Nair@lumina.example` and `priya.nair@lumina.example` are the same mailbox, and with citext the database agrees without any lower() gymnastics on every comparison.

Notice also what manager_id does: it points at the very table it lives in. An agent's manager is another agent. That one column gives the schema an org chart for free, and in Chapter 7 it will teach you self-joins, the technique of joining a table to itself.

Customers follow the same pattern, plus one column worth defending:

```
CREATE TABLE lumina.customer (
    id            bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    company_name  text NOT NULL,
    full_name     text NOT NULL,
    email         citext NOT NULL UNIQUE,
    plan          text NOT NULL,
    signed_up_at  timestamptz NOT NULL
);
```

*Listing 2.8 — Customers. The plan column earns its keep in every reporting chapter.*

Why is plan just text? Shouldn't it be restricted to free, pro, or enterprise? Yes, eventually, and PostgreSQL gives us three different tools for exactly that job. Choosing among them is a genuine design decision, so it gets a proper treatment in Chapters 5

and 6 rather than a hand-wave here. For two chapters, we trust ourselves to type plan names correctly. It will go fine, mostly, which is itself part of the lesson.

You may have noticed the types so far were picked with a certain confidence: `text` here, `bigint` there, `timestamptz` for every moment in time. That confidence is borrowed from Chapter 5, which spends its whole length on why these defaults are right and when to depart from them. Today we choose by feel; Chapter 5 brings the framework.

## The ticket and its conversation

Now the table everything else orbits. A ticket connects a customer to the team and agent handling their problem, carries the problem itself, and tracks a small life story: when it was created, when someone first responded, when it was resolved, when it was closed, and how the customer felt about the whole affair.

```
CREATE TABLE lumina.ticket (
    id                 bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    reference          text NOT NULL UNIQUE,
    customer_id        bigint NOT NULL,
    assigned_agent_id  bigint,
    team_id            bigint NOT NULL,
    subject            text NOT NULL,
    body               text NOT NULL,
    status             text NOT NULL,
    priority           text NOT NULL,
    channel            text NOT NULL,
    created_at         timestamptz NOT NULL,
    first_response_at  timestamptz,
    resolved_at        timestamptz,
    closed_at          timestamptz,
    satisfaction       smallint
);
```

*Listing 2.9 — The heart of the schema. Read the NULLs as carefully as the columns.*

The columns with NOT NULL are the facts a ticket cannot exist without. The columns without it tell the ticket's story by their absence. A ticket with no assigned_agent_id is sitting in a queue, waiting for a human. A NULL first_response_at is a customer still waiting to hear from anyone, which is precisely the number a support team gets judged on. A NULL resolved_at means work in progress; a NULL satisfaction means the survey went unanswered, which is not at all the same thing as a score of zero. We did not add

these nullable columns grudgingly. We designed the emptiness in, because the emptiness means something.

The `reference` column deserves a word. The id is for the database; the reference — LUM-1001, LUM-1002 — is for humans, printed in email subjects and read aloud on the phone. Keeping them separate means the database can renumber, partition, or shard by id someday without a single customer noticing. **UNIQUE** makes PostgreSQL the enforcer: two tickets can never share a reference, no matter which app, script, or hurried intern does the inserting.

The status column encodes the ticket's life as a small state machine, and it is worth spelling out because every reporting query in Part II leans on it. A ticket is born open. When an agent asks the customer a question and the ball leaves Lumina's court, it becomes waiting_on_customer. When the agent believes the problem is fixed it turns resolved, and after a quiet grace period, closed. The timestamps deliberately echo the lifecycle — yes, that is a controlled bit of duplication, and Chapter 6 will let the database police the agreement: resolved_at gets stamped at the third transition, closed_at at the fourth. Notice what did not get a column: there is no is_open boolean, no separate flags fighting with status about the truth. One column owns the lifecycle. When two columns can disagree about the same fact, sooner or later they will, and you are back to the spreadsheet problem wearing a nicer shirt.

Could someone set status to 'banana'? Today, yes. The state machine lives in our heads, and text columns believe anything they are told. Sit with that discomfort for four chapters. Chapter 5 weighs the tools for restricting a column to a known set of values, Chapter 6 installs the winner, and the discomfort is the reason you will care.

A ticket without conversation is just a complaint. The back-and-forth lives in its own table, one row per message:

```
CREATE TABLE lumina.ticket_comment (
    id                bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    ticket_id         bigint NOT NULL,
    parent_comment_id  bigint,
    author_kind       text NOT NULL,
    author_id         bigint NOT NULL,
    body              text NOT NULL,
    created_at        timestamptz NOT NULL
```

```
);
```

*Listing 2.10 — One row per message. parent_comment_id turns a list into a tree.*

Two design moves here. First, author_kind plus author_id records who spoke: the pair ('agent', 1) is Priya, ('customer', 2) is Jack Chen. A comment's author can come from either the agent table or the customer table, and this two-column pattern is the simplest honest way to say so. Second, parent_comment_id does for comments what manager_id did for agents: it points back into the same table. A comment can reply to a comment. Follow the chain and you get a thread; a thread is a tree; and walking trees is exactly what Chapter 9's recursive queries were born for.

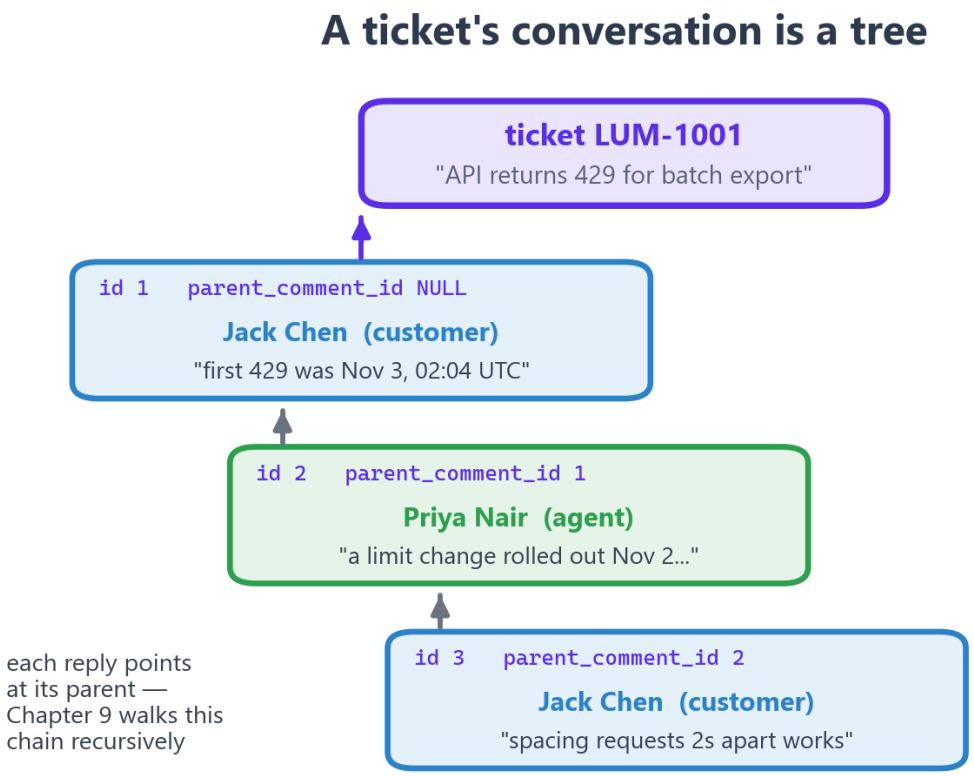


*Figure 2.3 — Ticket LUM-1001's thread. Each reply carries its parent's id; the chain is the conversation.*

One more table belongs to the ticket's inner circle. Things happen to a ticket — it gets created, assigned, escalated, resolved — and Lumina wants a permanent, append-only record of every one of those moments:

```
CREATE TABLE lumina.ticket_event (
    id           bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    ticket_id    bigint NOT NULL,
    event_kind   text NOT NULL,
    payload      jsonb NOT NULL DEFAULT '{}',
    occurred_at  timestamptz NOT NULL
);
```

*Listing 2.11 — The audit trail. Rows are only ever added, never edited.*

The payload column is our first sighting of `jsonb` — a whole JSON document stored in a single column. Why? Because an 'assigned' event wants to record which agent, a 'status_changed' event wants old and new values, and next quarter someone will invent an event we have not thought of. The shape of the data varies by row, and jsonb is built for exactly that. It is also a loaded weapon with a whole chapter of safety training attached; until Chapter 11, admire it from a distance.

This table also carries the chapter's only `DEFAULT`: a payload left unspecified becomes an empty document rather than NULL. That is a small kindness with a real payoff — code reading the audit trail can always ask the payload a question without first checking whether there is a payload to ask. An empty answer and a missing answerer are different things, and Chapter 4 will make a surprising amount of hay from that distinction.

Why give events their own table at all, when the ticket already has status columns? Because the ticket row stores the current truth and overwrites it freely, while the event table stores history and never overwrites anything. Those are different jobs with different rules. When a customer disputes what happened on their urgent ticket, the ticket row can only shrug about the present; the event trail answers with timestamps. Append-only tables like this also grow without limit, which is not today's problem — it is Chapter 19's problem, where this exact table gets carved into monthly slices while queries keep working untouched.

## Tags: when both sides can have many

Every relationship so far has been one-to-many. One team has many agents; one ticket has many comments. Tags break the pattern. A ticket about a billing crash carries both the billing tag and the crash tag — and each of those tags sits on dozens of other tickets. Many tickets, many tags, freely mixed. Where does that connection live? Not in either table. A column on ticket could hold one tag id, not many. The answer is a third table whose rows are the connections themselves:

```
CREATE TABLE lumina.tag (
    id           bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name         text NOT NULL UNIQUE,
    description  text NOT NULL
);

CREATE TABLE lumina.ticket_tag (
    ticket_id  bigint NOT NULL,
    tag_id     bigint NOT NULL,
    PRIMARY KEY (ticket_id, tag_id)
);
```

*Listing 2.12 — The join table: two columns, and each row is one act of tagging.*

ticket_tag has no id of its own, and that is the interesting part. Its primary key is the pair of columns together. Row (14, 12) means ticket 14 wears tag 12; the pair IS the fact, so the pair is the key, and the composite primary key makes duplicate tagging structurally impossible rather than merely discouraged. Tables like this are called join tables, and once you recognize the shape you will see it everywhere: students and courses, actors and films, users and roles. Same two-column skeleton every time.

## The rest of the cast

Two supporting tables finish the schema. Attachments — the screenshots and PDFs customers send — hang off a ticket, and optionally off the specific comment they arrived with. And SLA policies record Lumina's response-time promises per priority level, which Chapter 8 will turn into the dashboard query every support manager actually opens in the morning.

```
CREATE TABLE lumina.attachment (
    id           bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
```

```
    ticket_id     bigint NOT NULL,
    comment_id    bigint,
    filename      text NOT NULL,
    content_kind  text NOT NULL,
    byte_size     bigint NOT NULL,
    uploaded_at   timestamptz NOT NULL
);

CREATE TABLE lumina.sla_policy (
    id                      bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name                    text NOT NULL,
    applies_to_priority     text NOT NULL UNIQUE,
    first_response_minutes  integer NOT NULL,
    resolution_minutes      integer NOT NULL
);
```

*Listing 2.13 — Attachments and SLA policies complete the ten-table core.*

Ten tables. Ask psql for the roll call:

```
\dt lumina.*
```

```
               List of tables
 Schema |      Name      | Type  |  Owner
--------+----------------+-------+----------
 lumina | agent          | table | postgres
 lumina | attachment     | table | postgres
 lumina | customer       | table | postgres
 lumina | sla_policy     | table | postgres
 lumina | tag            | table | postgres
 lumina | team           | table | postgres
 lumina | ticket         | table | postgres
 lumina | ticket_comment | table | postgres
 lumina | ticket_event   | table | postgres
 lumina | ticket_tag     | table | postgres
(10 rows)
```

*Listing 2.14 — The complete Lumina core, straight from the catalog.*

And inspect the centerpiece with `\d`, which you met in Chapter 1 and will use daily for the rest of your career:

```
\d lumina.ticket
```

```
                                    Table "lumina.ticket"
      Column       |           Type           | Collation | Nullable |           Default
-------------------+--------------------------+-----------+----------+----------------------------
---
 id                | bigint                   |           | not null | generated always as
identity
 reference         | text                     |           | not null |
```

```
 customer_id       | bigint                   |           | not null |
 assigned_agent_id | bigint                   |           |          |
 team_id           | bigint                   |           | not null |
 subject           | text                     |           | not null |
 body              | text                     |           | not null |
 status            | text                     |           | not null |
 priority          | text                     |           | not null |
 channel           | text                     |           | not null |
 created_at        | timestamp with time zone |           | not null |
 first_response_at | timestamp with time zone |           |          |
 resolved_at       | timestamp with time zone |           |          |
 closed_at         | timestamp with time zone |           |          |
 satisfaction      | smallint                 |           |          |
Indexes:
    "ticket_pkey" PRIMARY KEY, btree (id)
    "ticket_reference_key" UNIQUE CONSTRAINT, btree (reference)
```

*Listing 2.16 — Every column, type, and constraint the ticket table carries so far.*

Read that output slowly once; you will skim it a thousand times after. The Nullable column is our design made visible: not null marks the facts a ticket cannot exist without, and the blanks mark the deliberate emptiness. The Default column shows identity generation on id and nothing anywhere else, which is honest — so far, the only value this table produces on its own is the key. At the bottom, psql lists the two indexes PostgreSQL created without being asked, one for the primary key and one enforcing the UNIQUE reference. We never typed CREATE INDEX. Constraints that promise uniqueness need a fast way to check it, so the index comes with the promise. Chapter 15 makes heavy use of that connection.

> **Pro Tip**
>
> *\d shows the shape; \d+ shows the shape plus storage details, compression, and per-column comments. The habit worth building: run \d on a table before you query it anywhere important. Ten seconds of reading beats twenty minutes of debugging a column that was never called what you assumed.*

Here is the whole schema in one picture. This diagram will reappear, in whole or in part, in nearly every chapter that follows — dog-ear the page.

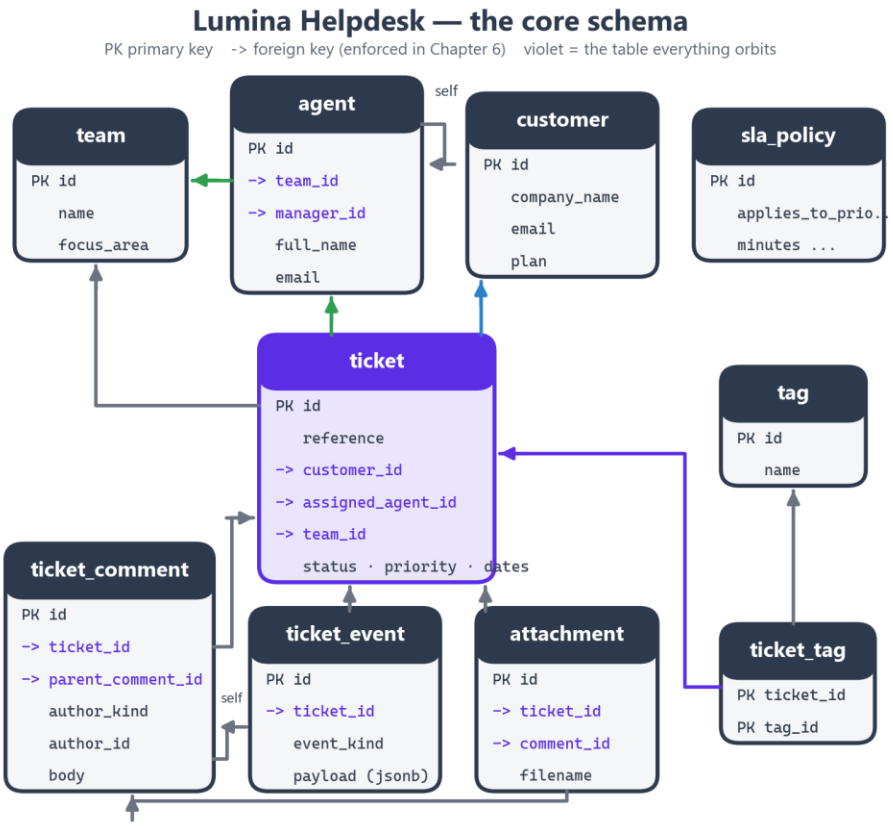


*Figure 2.4 — The Lumina Helpdesk core. Arrows show which columns point where; Chapter 6 makes the arrows enforceable.*

### Going Deeper

*What did CREATE TABLE actually create? On disk, a PostgreSQL table is a heap: a plain file of fixed-size 8 KB pages, each holding as many rows as fit, in no particular order. There is no magic in the file — the magic is in everything the server layers on top: caching, concurrency, crash recovery, and the indexes we will add in Chapter 15. When Chapter 18 talks about tables getting bloated, it is these pages filling with dead space. For now, one honest sentence is enough: a table is a file of pages, and every design decision downstream respects that fact.*

## What we deliberately left out

A confession before you get attached to this schema: it is unarmored. Nothing stops you from inserting a ticket whose customer_id is 999999, a customer who does not exist. Nothing stops a satisfaction score of 47, or a status of 'banana'. The columns that point between tables are, so far, just numbers that happen to agree.

This is a choice, not an oversight. The armor — foreign keys, CHECK constraints, and their relatives — deserves its own chapter, and more importantly, you will learn far more adding constraints to a live, populated database in Chapter 6 than you would copying them blindly today. That is how it happens in real projects anyway: schemas are born optimistic and get defensive with age. Chapter 3 will even leave a small mess behind on purpose, and cleaning it up is what makes Chapter 6 stick.

> **Cross-Reference**
>
> *Types get their full treatment in Chapter 5, including why plan is still plain text. Constraints arrive in Chapter 6, where every arrow in Figure 2.4 becomes a rule the database enforces. The companion repository's schema/010-core-schema.sql holds exactly the DDL from this chapter, and reset-to-chapter.ps1 -Chapter 3 will apply it plus the seed data waiting for you in the next chapter.*

> **Try It**
>
> *Before Chapter 3: sketch the ERD for a domain you know well — a library, a gym, a football league. Circle the nouns, draw the arrows, and find at least one many-to-many that needs a join table. Ten minutes with a pencil now will make the next twenty chapters feel familiar.*

## Summary

You started with a spreadsheet that photocopied the same facts onto every row, and you watched that repetition turn into contradictions waiting to happen. The fix was the founding move of relational design: one table per kind of thing, one home per fact, and plain numeric columns connecting them. Along the way you placed those tables in PostgreSQL's hierarchy — cluster, database, schema, table — and put search_path to work so Lumina's tables answer to short names.

The schema you built says more than its column list. Identity columns hand out keys without your help; UNIQUE guards emails and ticket references; nullable columns carry meaning in their emptiness; two self-referencing columns quietly hold an org chart and threaded conversations; and a composite-key join table makes duplicate tagging impossible by construction. You also know what is missing — the constraints that will make the arrows between tables enforceable — and that the gap is deliberate, with Chapter 6 waiting to close it.

## Key Terms

- **Entity** — a kind of thing the domain cares about; each entity becomes a table.
- **Normalization** — splitting data so each fact is stored exactly once; the cure for update anomalies.
- **Schema** — a named namespace for tables inside a database.
- **search_path** — the ordered list of schemas PostgreSQL checks to resolve unqualified names.
- **Primary key** — the column (or columns) that uniquely identify each row.
- **Identity column** — a key column whose values PostgreSQL generates itself (GENERATED ALWAYS AS IDENTITY).
- **Composite key** — a primary key built from more than one column, as in ticket_tag.
- **Join table** — a two-column table whose rows record a many-to-many relationship.
- **Self-reference** — a column pointing at its own table: manager_id, parent_comment_id.

## Practice Exercises

### Exercise 2.1 — Basic: A table of your own

The book added a tag table for you, but suppose it had not. Write CREATE TABLE for a tag entity from scratch: a generated id, a unique name, a description. Then create it under a scratch name (tag_practice), inspect it with \d, and drop it.

*Hint:* copy the shape of listing 02-12 and change what needs changing. DROP TABLE cleans up after you.

### Exercise 2.2 — Intermediate: The knowledge base

Lumina's agents keep solutions in a wiki nobody can search. Sketch a kb_article table: written by an agent, optionally born from a ticket, with a title, a body, and a creation time. Decide which columns are NOT NULL and defend each decision in a sentence. Keep your DDL — Chapter 23 builds semantic search on a table that starts here.

*Hint:* "optionally born from a ticket" is doing a lot of work in that sentence. Which column, and which constraint is conspicuously absent?

### Exercise 2.3 — Challenge: Break the spreadsheet

Return to helpdesk_sheet from listing 02-01 and play saboteur, on paper. Fiona changes her email address. List every cell that must change, then suppose one row spells her name "Fiona OBrien" (no apostrophe) and explain, in two sentences, which cells any name-based correction will silently miss and what the sheet claims about her afterward. Close with one sentence on why the normalized customer table makes that particular corruption impossible by construction. No SQL required — Chapter 3 hands you the verbs, and this exercise becomes a two-line demonstration you may enjoy revisiting.

*Hint:* the corruption is not in the cells that change. It is in the row that gets missed — and in the normalized schema, there is exactly one cell that could ever need changing.

## What's Next

An empty schema is a stage with no actors. Chapter 3 teaches the four verbs every application speaks — INSERT, SELECT, UPDATE, DELETE — and uses them to seed Lumina with the cast you have already met: three teams, six agents, twelve customers, and forty tickets whose stories will power every example through the rest of the book. You will also meet RETURNING, a small PostgreSQL kindness that ORM users spend years not knowing they are missing. Bring the schema; the data is next.

# End of the Sample

You have read two of twenty-three chapters. Chapter 1 got a server running; Chapter 2 turned a spreadsheet nobody could search into the Lumina Helpdesk schema, with a deliberate gap where the constraints will go. That gap is the hinge — everything after this point is the database earning its keep.

## What the complete book covers

- **Relational thinking** — constraints that make bad data impossible, joins across five tables, aggregation, CTEs, and window functions.
- **What makes PostgreSQL PostgreSQL** — JSONB beside relational columns, full-text search, server-side functions and triggers, and MVCC without hand-waving.
- **Performance and operations** — indexes, EXPLAIN plans you can act on, row-level security, backups that restore, partitioning, and replication.
- **PostgreSQL from your code** — Npgsql, EF Core, and pgvector for semantic and hybrid search in the database you already run.

Every listing in those chapters ran against a live PostgreSQL 18.4 before it went to print, and the output beneath it was captured from that run — including the EXPLAIN plans.

> **Key Takeaway**
>
> *The complete book — twenty-three chapters, five appendices, and a page-numbered index — is on Leanpub in EPUB and PDF, DRM-free.*

**Get the complete book**

**leanpub.com/thepostgresqlhandbook**

The companion code is free either way, MIT-licensed:
`github.com/RachidD68/the-postgresql-handbook`