claim query · jobs.py

```
UPDATE jobs SET locked_at = now()
WHERE id = (
  SELECT id FROM jobs
  WHERE done_at IS NULL
  ORDER BY run_at, id
  FOR UPDATE SKIP LOCKED
  LIMIT 1)
RETURNING id, kind, payload;

-- JSONB · search · queues · RLS
```

JSONB

EXPLAIN

SKIP LOCKED

RLS

```
=> SELECT * FROM course_stats; 5 rows
=> EXPLAIN (ANALYZE) SELECT ... ORDER BY id
=> LISTEN larkhaven_events;
```

A HANDS-ON GUIDE FOR DEVELOPERS

PostgreSQL for App Developers

Design, query, and ship data-driven applications with PostgreSQL 18

Schemas · JSONB · search · pagination · job queues
concurrency · migrations · EXPLAIN · pooling · RLS

DESIGN · QUERY · SCALE · SHIP

THENEXBOOKS

PostgreSQL for App Developers

Design, Query, and Ship Data-Driven Applications with PostgreSQL 18
Schemas, queries, search, job queues, concurrency, migrations, testing, and performance — from application code

First Edition

THENEXBOOKS

PostgreSQL for App Developers
Design, Query, and Ship Data-Driven Applications with PostgreSQL 18

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. 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. PostgreSQL, its drivers, and the tools discussed in this book evolve; always consult the official documentation for the versions you use.

PostgreSQL is a registered trademark of the PostgreSQL Community Association. Python is a trademark of the Python Software Foundation. Docker is a trademark of Docker, Inc. Amazon RDS and Amazon Aurora are trademarks of Amazon.com, Inc. or its affiliates; Google Cloud SQL is a trademark of Google LLC; Microsoft Azure is a trademark of Microsoft Corporation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the PostgreSQL Global Development Group.

Larkhaven Learning, the business used as the running example in this book, is fictional, as are its schools, people, courses, learners, and data. Domains such as example.com and larkhaven.example are reserved for examples.

Preface

Most applications are, at heart, a careful conversation with a database. Every sign-up, purchase, search box, dashboard, and background email depends on queries that must be correct when two users click at the same moment, fast when the table holds millions of rows, and safe to change while the app keeps running. PostgreSQL is exceptionally good at all of this — if you know what it offers and how to use it from your code.

Many developers treat PostgreSQL as a place to store rows behind an ORM. They miss out. PostgreSQL can enforce your business rules with constraints, return exactly the JSON your API needs, search text with ranking and typo tolerance, run a reliable job queue, notify your app of changes, isolate tenants with row-level security, and explain precisely why a query is slow. Used well, it removes whole categories of bugs and whole pieces of infrastructure from your system.

This book is written for application developers. Every chapter starts from a feature or a problem a real team meets — a dashboard that double counts, pagination that skips rows, two workers taking the same job, a migration that freezes the site — and shows the PostgreSQL technique that solves it, the SQL, and the application code around it. The SQL examples were run on PostgreSQL 18 while writing, and the output you see is real.

Who This Book Is For

- **Backend and full-stack developers** who use PostgreSQL every day and want to use it well rather than just get by.
- **Developers moving from MySQL, SQL Server, SQLite, or a NoSQL store** who want to learn what PostgreSQL does differently and what it does better.
- **ORM users** who want to understand the SQL their tools generate, fix N+1 queries, and know when to write SQL directly.
- **Tech leads and students** who want a practical reference for building correct, fast, maintainable data layers.

You should know basic SQL (SELECT, INSERT, joins) and one programming language. Application code in this book is Python with psycopg 3, written so that developers using Node.js, Java, Go, or another language can follow along; Appendix B maps the driver calls to those languages.

What You Will Learn

- How to design application schemas with the right keys, constraints, and data types — and the type mistakes that cause bugs with money and time zones.
- How to talk to PostgreSQL from code: parameters, transactions, row factories, bulk loading, and connection pooling with psycopg 3.

- CRUD done right with RETURNING, upserts, and MERGE; reporting queries without double counting; keyset pagination; CTEs, LATERAL, and window functions.
- JSONB, arrays, ranges, and exclusion constraints; full-text search with ranking and highlighting; fuzzy search with trigrams.
- Job queues with FOR UPDATE SKIP LOCKED, LISTEN/NOTIFY, triggers, and the transactional outbox.
- Transactions and isolation levels from the application's point of view; lost updates, optimistic locking, advisory locks, and deadlocks.
- Constraints as a safety net and turning database errors into clean API responses.
- Zero-downtime schema migrations and testing database code with real PostgreSQL.
- Reading EXPLAIN, choosing indexes, fixing N+1 queries in ORMs, and configuring pools and timeouts.
- Multi-tenancy with row-level security, least-privilege roles, and running PostgreSQL in production.

How This Book Is Organized

Part I — Foundations for App Developers (Chapters 1–5) sets up your environment, designs the schema, covers the data types developers get wrong, and connects from code. **Part II — Querying for Features** (Chapters 6–10) covers writes, reporting, pagination, advanced queries, and JSONB. **Part III — Search, Events, and Background Work** (Chapters 11–15) covers search, arrays and ranges, job queues, notifications and the outbox, and when logic belongs in the database. **Part IV — Correctness and Concurrency** (Chapters 16–20) covers transactions, concurrency bugs, constraints and error handling, migrations, and testing. **Part V — Performance and Scale** (Chapters 21–25) covers EXPLAIN, indexes, ORMs, pooling, and security. **Part VI — Projects and Production** (Chapters 26–28) assembles the complete Larkhaven data layer and background system and prepares you to run it in production. The appendices provide a SQL and psql cheat sheet, a driver reference across languages, an error-code guide, and a glossary.

Each chapter ends with practice exercises, selected solutions, and a **Key Takeaways** box for quick review.

The Running Example: Larkhaven Learning

Throughout the book we build the data layer of **Larkhaven Learning**, a fictional online course platform where independent schools publish courses and learners enroll, follow lessons, track their progress, and leave reviews. Maya, the founder, keeps asking for features; Omar, the backend lead, keeps spotting what will break in production; and Lin, a new developer on the team, is learning PostgreSQL alongside you. A small, fixed set of sample data — two schools, five learners, five courses — runs through every chapter, so you can check each result yourself.

Versions and Conventions

Examples target **PostgreSQL 18**; most SQL also runs on PostgreSQL 15 and later, and features that need a newer release are labeled. Application code uses Python 3.12, psycopg 3, psycopg_pool, pytest, and SQLAlchemy 2.0. SQL output shown in this book comes from real runs; output from application code is shown as what you should see, because environments differ. Where a detail may change, a Note points you to the official documentation rather than guessing.

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

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

> **WARNING**
> A common mistake or something that can cause bugs, data loss, or security problems.

Code appears in `monospace`. SQL keywords are written in uppercase. Long lines are wrapped to fit the page. Captions above code blocks, such as `larkhaven/jobs.py` or `psql`, tell you where the code lives or where to run it.

Contents

Preface

PART I — FOUNDATIONS FOR APP DEVELOPERS

1 Why PostgreSQL for Applications: What You Get Beyond "Just a Database"

2 A Developer Setup: PostgreSQL 18, psql, Docker, and a Python Project

3 Designing the Schema: Tables, Keys, Constraints, and Naming for Applications

4 Data Types Developers Get Wrong: Money, Time Zones, Text, UUIDs, and Enums

5 Connecting from Code: psycopg 3, Parameters, Transactions, and Row Factories

PART II — QUERYING FOR FEATURES

6 CRUD Done Right: RETURNING, Upserts with ON CONFLICT, and MERGE

7 Joins, Aggregates, and Reporting Queries for Dashboards

8 Pagination, Sorting, and Filtering: OFFSET vs Keyset and Dynamic Filters

9 CTEs, LATERAL, and Window Functions for Real Features

10 JSONB for Flexible Data: When to Use It, Querying, and Indexing

PART III — SEARCH, EVENTS, AND BACKGROUND WORK

11 Full-Text Search and Fuzzy Matching: tsvector, Ranking, Highlighting, and pg_trgm

12 Arrays, Ranges, and Exclusion Constraints: Tags and Scheduling

13 Job Queues in PostgreSQL with FOR UPDATE SKIP LOCKED

14 LISTEN/NOTIFY, Triggers, and the Transactional Outbox

15 Views, Functions, and Generated Columns: When Logic Belongs in the Database

PART IV — CORRECTNESS AND CONCURRENCY

16 Transactions and Isolation Levels from the Application's Point of View

17 Concurrency Bugs and Their Fixes: Lost Updates, Optimistic Locking, and Advisory Locks

18 Constraints as a Safety Net: Integrity Rules, Deferrable Constraints, and Handling Errors in Code

19 Schema Migrations Without Downtime: Tools, Patterns, and Expand/Contract

20 Testing Database Code: Fixtures, Rollback Isolation, Testcontainers, and pgTAP

PART V — PERFORMANCE AND SCALE

21 Reading EXPLAIN as a Developer

22 Indexes for Application Queries: Composite, Partial, Expression, Covering, and GIN

23 ORMs, N+1 Queries, and Query Patterns That Scale (SQLAlchemy 2.0)

24 Connection Pooling, Timeouts, and Resilience: psycopg_pool and PgBouncer

25 Multi-Tenancy and Security: Roles, Row-Level Security, and Secrets

PART VI — PROJECTS AND PRODUCTION

26 Project: The Larkhaven Data Layer End to End

27 Project: Search, Background Jobs, and Notifications for Larkhaven

28 Shipping to Production: Managed PostgreSQL, Backups, Observability, Upgrades, and Next Steps

APPENDICES

A SQL and psql Cheat Sheet for App Developers

B psycopg 3 Reference with Node.js, Java, and Go Equivalents

C Common PostgreSQL Errors (SQLSTATE) and What to Do in Code

D Glossary

PART I

Foundations for App Developers

What PostgreSQL gives an application beyond storing rows, a productive developer setup, a schema designed for a real app, the data types developers get wrong, and talking to PostgreSQL safely from code.

1 · Why PostgreSQL for Applications
2 · A Developer Setup
3 · Designing the Schema
4 · Data Types Developers Get Wrong
5 · Connecting from Code

CHAPTER 1 · PART I — FOUNDATIONS FOR APP DEVELOPERS

Why PostgreSQL for Applications: What You Get Beyond "Just a Database"

Many applications use PostgreSQL as a place to put rows and get them back. That works, but it leaves most of the value on the table: a server that can enforce your business rules, coordinate your workers, search your content, and isolate your tenants — if you let it.

A Database That Does More Than Store Rows

If you have built a web application before, you have probably met the "dumb storage" style of using a database. The framework creates some tables, every rule lives in application code, and the database is treated as a slightly slow dictionary. Validation happens in a form class. Uniqueness is checked with a `SELECT` before an `INSERT`. Background work goes to a separate queue service. Search goes to a separate search engine. Each of those choices adds code, infrastructure, and failure modes.

PostgreSQL can carry much of that weight itself. It is a relational database and also a rule engine (constraints), a coordination service (transactions and row locks), a document store (JSONB), a search engine (full-text search plus trigram matching), a message bus for small payloads (`LISTEN` / `NOTIFY`), and an access-control layer (roles and row-level security). None of these replace dedicated tools at every scale, but for most applications they are more than enough, and they have a property no external service can offer: they participate in the same transaction as your data.

That last point is the theme of this book. When the rule, the queue, and the data live in one transactional system, a whole class of bugs disappears. You cannot enqueue a welcome email for an enrollment that was rolled back. You cannot have two users with the same email address because two requests raced each other. You cannot accidentally show one school's draft courses to another school. This chapter is a guided tour of those capabilities, each with a tiny real example, followed by just enough of PostgreSQL's architecture to reason about it as an application developer.

Meet Larkhaven Learning

Every example in this book uses one application: **Larkhaven Learning**, a fictional online course platform. Independent **schools** publish **courses** made of **lessons**. Learners (**users**) **enroll** in courses, mark lessons complete, and leave **reviews**. Some courses run **live sessions** at fixed times. Behind the scenes, the platform sends emails and generates certificates through a **job queue**, and publishes domain events through an **outbox**.

Three people will appear throughout the book:

- **Maya**, the founder, asks product questions: "Can we show a progress bar?", "Can learners search for courses?", "How much revenue did each school make?"
- **Omar**, the backend lead, reviews code and handles production issues. He is the one who notices the race condition, the slow query, or the missing index.
- **Lin**, a new developer, is learning PostgreSQL alongside you. Lin asks the questions you are probably asking, and occasionally writes the bug we then fix together.

The sample data is small on purpose. There are two schools (Brushwork Art School and Byteforge Academy), five users (Ana, Ben, Chen, Dana, and Eli), five courses (four published and one draft), fourteen lessons, seven enrollments, and six reviews. Small data means you can check every result by eye. Chapter 2 shows how to load this data on your machine; for now, read the examples and trust that every output in this book came from a real run on PostgreSQL 18.

The Tour: Eight Things You Get for Free

Each subsection below shows one capability with one or two statements against the Larkhaven database. Every feature here gets its own chapter later. The goal is to build a mental list of "the database can do that" so you reach for it at the right moment.

1. Constraints: rules that cannot be bypassed

A constraint is a rule the database checks on every write, no matter which code path performed it: your API, an admin script, a migration, or a developer in psql at midnight. Larkhaven's `courses` table says a price can never be negative, the `users` table has a unique index on `lower(email)`, and enrollments must point at real users. Watch all three refuse bad data:

psql

```
larkhaven=> INSERT INTO courses (school_id, slug, title, price_cents)
larkhaven-> VALUES (2, 'free-money', 'Free Money', -500);
ERROR:  new row for relation "courses" violates check constraint
"courses_price_cents_check"
DETAIL:  Failing row contains (6, 2, free-money, Free Money, , -500, USD, draft, ...).
larkhaven=> INSERT INTO users (email, display_name)
larkhaven-> VALUES ('Ana@Example.com', 'Ana Again');
ERROR:  duplicate key value violates unique constraint "users_email_key"
DETAIL:  Key (lower(email))=(ana@example.com) already exists.
larkhaven=> INSERT INTO enrollments (user_id, course_id, price_paid_cents)
larkhaven-> VALUES (99, 1, 2900);
ERROR:  insert or update on table "enrollments" violates foreign key constraint
"enrollments_user_id_fkey"
DETAIL:  Key (user_id)=(99) is not present in table "users".
```

(We trimmed and wrapped two long lines to fit the page.) Notice the second error. The existing row is `ana@example.com`; the new one is `Ana@Example.com`. A plain `UNIQUE (email)` would have accepted both, because the strings differ. The index on the expression `lower(email)` enforces uniqueness the way

humans think about email addresses. An application-side check ("SELECT first, then INSERT") cannot do this reliably: two sign-up requests arriving in the same millisecond can both see "no such user" and both insert. The unique index makes the second one fail, every time.

Reading the error

PostgreSQL error messages are structured, and your application code can use that structure. Each error has a five-character SQLSTATE code (`23514` for a check violation, `23505` for a unique violation, `23503` for a foreign key violation), the constraint name, and a `DETAIL` line describing the offending values. psycopg 3 turns these into exception classes such as `psycopg.errors.UniqueViolation`, so the enrollment code in later chapters can catch a unique violation and return a friendly "you are already enrolled" message instead of a 500 error. Good constraint names pay off here: `users_email_key` tells the reader of a log line exactly what went wrong. Chapter 3 covers naming, and Chapter 18 covers mapping errors to application behavior.

> **NOTE**
>
> The failing `courses` row shows `id` 6 even though nothing was inserted. Identity values come from a sequence, and a sequence value used by a failed or rolled-back insert is never handed out again. Gaps in ids are normal and harmless; never write application logic that expects ids to be consecutive.

2. Transactions: all or nothing

When Ana enrolls in a course, Larkhaven needs to do several things: insert the enrollment, record an event in the outbox, and queue a welcome email. If any step fails, none of them should be visible. A transaction gives you exactly that:

psql

```
larkhaven=> BEGIN;
BEGIN
larkhaven=*> INSERT INTO enrollments (user_id, course_id, price_paid_cents)
larkhaven-*> VALUES (5, 3, 1900) RETURNING id, user_id, course_id;
 id | user_id | course_id
----+---------+-----------
  8 |       5 |         3
(1 row)

INSERT 0 1
larkhaven=*> INSERT INTO outbox (topic, payload)
larkhaven-*> VALUES ('enrollment.created',
larkhaven(*>         '{"enrollment_id": 8, "user_id": 5, "course_id": 3}');
INSERT 0 1
larkhaven=*> ROLLBACK;
ROLLBACK
larkhaven=> SELECT count(*) FROM enrollments WHERE user_id = 5;
 count
```

```
-------
     0
(1 row)
```

The asterisk in the prompt (`larkhaven=*>`) is psql telling you a transaction is open. After `ROLLBACK`, Eli has no enrollments and the outbox row is gone too. Had we written `COMMIT`, both rows would have become visible to everyone at the same instant. Other sessions never see a half-finished enrollment. Chapter 5 shows how psycopg 3 maps this onto `with conn.transaction():` blocks, and Chapter 16 explains isolation levels — what concurrent transactions can and cannot see of each other.

3. Rich types: JSONB, arrays, and ranges

PostgreSQL's type system goes far beyond numbers and strings. Three types matter most for applications.

JSONB stores structured documents in a binary format that can be indexed and queried. Larkhaven keeps lesson content as a list of blocks, because a lesson might contain text, a video, and a code sample in any order. The containment operator `@>` finds lessons that contain at least one video block:

psql

```
larkhaven=> SELECT id, title, jsonb_array_length(body->'blocks') AS blocks
larkhaven-> FROM lessons
larkhaven-> WHERE body @> '{"blocks": [{"type": "video"}]}';
 id |       title       | blocks
----+-------------------+--------
  2 | Your First SELECT |      2
(1 row)
```

Arrays hold a list of values in one column. Courses carry tags as `text[]`, and a GIN index makes "courses tagged beginner" fast:

psql

```
larkhaven=> SELECT slug, tags FROM courses WHERE tags @> '{beginner}' ORDER BY id;
      slug      |      tags
----------------+----------------
 sql-basics     | {sql,beginner}
 watercolor-101 | {art,beginner}
(2 rows)
```

Ranges represent an interval as one value, with operators for overlap (`&&`), containment, and adjacency. Live sessions use a `tstzrange`, and an exclusion constraint guarantees that two sessions of the same course never overlap. Course 5 already has a session from 16:00 to 17:00 UTC on April 14. Try to book another one that starts at 16:30:

psql

```
larkhaven=> INSERT INTO live_sessions (course_id, title, during)
larkhaven-> VALUES (5, 'Extra Q&A', '[2026-04-14 16:30Z, 2026-04-14 17:30Z)');
ERROR:  conflicting key value violates exclusion constraint
"live_sessions_course_id_during_excl"
DETAIL:  Key (course_id, during)=(5, ["2026-04-14 16:30:00+00","2026-04-14 17:30:00+00"))
conflicts with existing key (course_id, during)=(5, ["2026-04-14 16:00:00+00",...)).
```

Double-booking prevention is notoriously hard to get right in application code, because the check and the insert race each other just like the email check did. Here it is one line in the table definition. Chapter 10 covers JSONB in depth and Chapter 12 covers arrays, ranges, and exclusion constraints.

4. Full-text search

Maya wants a search box. Before reaching for a separate search service, look at what PostgreSQL already provides. The `courses` table has a generated `search` column of type `tsvector`, which stores stemmed words from the title (weight A) and summary (weight B), plus a GIN index on it. A query takes the same kind of input users type into a search box:

psql

```
larkhaven=> SELECT slug, ts_rank(search, q) AS rank
larkhaven-> FROM courses, websearch_to_tsquery('english', 'sql') AS q
larkhaven-> WHERE search @@ q;
    slug    |   rank
------------+-----------
 sql-basics | 0.6079271
(1 row)
```

Only "SQL Basics for Beginners" matches. "PostgreSQL for App Developers" does not, because full-text search works on whole words, and "postgresql" is a different word from "sql". That is often what you want, and sometimes not — which is why Chapter 11 pairs full-text search with fuzzy matching from the `pg_trgm` extension, which we meet in a moment.

5. Job queues with SKIP LOCKED

Background work — sending emails, generating certificates, reindexing content — usually needs a queue. PostgreSQL can be that queue. The trick is `FOR UPDATE SKIP LOCKED`: a worker locks the next available job row, and other workers *skip* locked rows instead of waiting for them. We queued two welcome-email jobs, then ran the heart of the claiming query from two terminals at nearly the same time. Terminal 1 keeps its transaction open for the moment:

psql — terminal 1

```
larkhaven=> BEGIN;
BEGIN
larkhaven=*> SELECT id, kind, payload FROM jobs
larkhaven-*> WHERE done_at IS NULL AND locked_at IS NULL AND run_at <= now()
larkhaven-*> ORDER BY run_at, id
```

```
larkhaven-*> FOR UPDATE SKIP LOCKED
larkhaven-*> LIMIT 1;
 id |        kind        |       payload
----+--------------------+---------------------
  1 | send_welcome_email | {"enrollment_id": 1}
(1 row)
```

psql — terminal 2 (while terminal 1 is still open)

```
larkhaven=> BEGIN;
BEGIN
larkhaven=*> SELECT id, kind, payload FROM jobs
larkhaven-*> WHERE done_at IS NULL AND locked_at IS NULL AND run_at <= now()
larkhaven-*> ORDER BY run_at, id
larkhaven-*> FOR UPDATE SKIP LOCKED
larkhaven-*> LIMIT 1;
 id |        kind        |       payload
----+--------------------+---------------------
  2 | send_welcome_email | {"enrollment_id": 2}
(1 row)
```

Terminal 2 did not block and did not get the same job. Two workers, two different jobs, no duplicate emails, no external broker. Because the queue is a table, enqueueing a job can happen in the same transaction as the enrollment that caused it: if the enrollment rolls back, the job never existed. Chapter 13 builds Larkhaven's complete queue with retries, backoff, and dead jobs; there the `SELECT` is wrapped in an `UPDATE` that marks the job as claimed and commits at once, so no transaction stays open while the email is sent.

6. LISTEN and NOTIFY

Polling a table every second is wasteful. PostgreSQL has a built-in publish/subscribe channel: a session runs `LISTEN channel`, and any session can send `NOTIFY channel, 'payload'`. Notifications are delivered only when the sending transaction commits, so listeners never hear about work that was rolled back. In a single psql session you can listen and notify yourself (the notification line is wrapped to fit the page):

psql

```
larkhaven=> LISTEN larkhaven_events;
LISTEN
larkhaven=> NOTIFY larkhaven_events, '{"topic": "enrollment.created", "id": 8}';
NOTIFY
Asynchronous notification "larkhaven_events" with payload "{"topic":
"enrollment.created", "id": 8}" received from server process with PID 54724.
```

Larkhaven's workers will use this to wake up immediately when a job arrives, while still polling occasionally as a safety net. Notifications are not stored: a listener that is disconnected misses them. That is why Chapter 14 combines `NOTIFY` with the transactional outbox table, which *is* durable.

7. Extensions

PostgreSQL is designed to be extended with new types, functions, operators, and index methods, packaged as extensions. Larkhaven uses two that ship with the standard PostgreSQL distribution: `pg_trgm` (similarity search on character trigrams) and `btree_gist` (lets the exclusion constraint above combine an ordinary `bigint` equality check with a range overlap check):

psql

```
larkhaven=> SELECT extname, extversion FROM pg_extension ORDER BY extname;
  extname   | extversion
------------+------------
 btree_gist | 1.8
 pg_trgm    | 1.6
 plpgsql    | 1.0
(3 rows)

larkhaven=> SELECT slug, round(word_similarity('postgres', title)::numeric, 2) AS score
larkhaven-> FROM courses
larkhaven-> ORDER BY score DESC
larkhaven-> LIMIT 2;
       slug       | score
------------------+-------
 postgres-apps    |  0.89
 portrait-drawing |  0.22
(2 rows)
```

Trigram similarity finds "PostgreSQL for App Developers" from the partial word "postgres", exactly the case full-text search missed. Many other extensions exist — PostGIS for geospatial data and pgvector for vector similarity are two well-known ones — but managed providers decide which they allow, so always check before you design around one. Chapter 28 returns to that question.

8. Row-level security

Larkhaven is multi-tenant: each school should see only its own courses in the school dashboard. You can enforce that with `WHERE school_id = ...` in every query, and one day someone will forget it. Row-level security (RLS) moves the filter into the database, attached to the table. Here is a minimal policy, tried as the application's login role `larkhaven_app`:

psql

```
larkhaven=> ALTER TABLE courses ENABLE ROW LEVEL SECURITY;
ALTER TABLE
larkhaven=> CREATE POLICY courses_tenant ON courses
larkhaven->   USING (school_id =
larkhaven(>          nullif(current_setting('app.school_id', true), '')::bigint);
CREATE POLICY
larkhaven=> SET ROLE larkhaven_app;
SET
larkhaven=> SELECT count(*) FROM courses;
 count
-------
```

```
     0
(1 row)

larkhaven=> BEGIN;
BEGIN
larkhaven=*> SELECT set_config('app.school_id', '1', true);
 set_config
------------
 1
(1 row)

larkhaven=*> SELECT id, slug, school_id FROM courses ORDER BY id;
 id |       slug       | school_id
----+------------------+-----------
  3 | watercolor-101   |         1
  4 | portrait-drawing |         1
(2 rows)

larkhaven=*> COMMIT;
COMMIT
larkhaven=> SELECT count(*) FROM courses;
 count
-------
     0
(1 row)
```

With no school set, the query sees nothing — a safe default. Inside a transaction that sets `app.school_id` to 1, it sees only Brushwork Art School's two courses, even though the query has no `WHERE` clause. The third argument `true` makes the setting local to the transaction, so it cannot leak into the next request that reuses this pooled connection.

> **WARNING**
>
> Superusers and table owners bypass RLS by default, so testing a policy while connected as the superuser or the table owner shows every row and proves nothing. Test as the role your application actually uses. The `nullif` handles a sharp edge visible right after the `COMMIT` above: once a transaction-local setting has been used in a session, `current_setting` returns an empty string rather than NULL afterwards, and a bare `''::bigint` cast would raise an error. `nullif` turns the empty string into NULL, so the last query safely sees nothing again. Chapter 25 builds the production version of this policy, including a second policy that lets anyone browse published courses.

How PostgreSQL Works, for App Developers

You do not need to know PostgreSQL's internals to build good applications, but three facts explain most of the behavior you will see in production.

Client and server

PostgreSQL is a server. Your application, psql, and GUI tools are all clients that open a network connection (TCP, usually port 5432, or a Unix-domain socket on the same machine), authenticate as a role, and send SQL over PostgreSQL's wire protocol. Nearly every client library — psycopg for Python, and many drivers for other languages — is built on `libpq`, the C client library, or reimplements its behavior. That is why the same connection strings and environment variables (`PGHOST`, `PGUSER`, and friends, covered in Chapter 2) work everywhere.

One process per connection

Each client connection is served by its own operating-system process on the server, called a backend. You can see yours:

psql

```
larkhaven=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
           6564
(1 row)

larkhaven=> SELECT pid, backend_type FROM pg_stat_activity
larkhaven-> WHERE backend_type <> 'client backend' OR pid = pg_backend_pid()
larkhaven-> ORDER BY pid;
  pid  |         backend_type
-------+------------------------------
  6564 | client backend
 36156 | checkpointer
 39680 | walwriter
 51136 | autovacuum launcher
 53844 | background writer
 58780 | io worker
 59076 | io worker
 59084 | io worker
 59236 | logical replication launcher
(9 rows)
```

The filter hides other client sessions, so the one `client backend` is our psql session; everything else is a background process the server runs for itself (the `io worker` processes are new in PostgreSQL 18, which performs some reads asynchronously). Your process ids will differ. The practical consequence of one process per connection is that connections are relatively expensive. Opening one takes a process start plus authentication, and each idle connection still holds memory on the server. A web application that opens a new connection per request, or one that runs 50 instances each holding 20 connections, will hit trouble long before the queries themselves are slow. The answer is a connection pool, either in your application (`psycopg_pool`) or in front of the server (PgBouncer). Chapter 24 is devoted to it.

MVCC in one paragraph

PostgreSQL uses multi-version concurrency control (MVCC). An `UPDATE` does not overwrite a row in place; it writes a new version of the row and marks the old version as expired by the updating transaction. Each transaction sees a snapshot: the row versions committed before its snapshot was taken. Consequences for you: readers never block writers and writers never block readers, so a long report does not freeze the checkout page; two writers updating the *same row* do wait for each other, which is how the `SKIP LOCKED` queue works; old row versions are cleaned up later by a background process called autovacuum, which is why a transaction left open for hours (an idle connection that never committed) is harmful — it forces the server to keep old versions alive. If you remember "short transactions, always commit or roll back", you have captured most of what MVCC asks of an application developer.

Where the Database Ends and the App Begins

"Put logic in the database" can be taken too far. Stored procedures that implement your entire business layer are hard to test, hard to deploy, and hard for new developers to find. The opposite extreme — treating PostgreSQL as dumb storage — throws away the guarantees we just toured. This book uses a simple rule of thumb:

Belongs in the database	Belongs in the application
Invariants that must never be broken: NOT NULL, unique, foreign keys, CHECK ranges	Workflow and orchestration: what happens after a user clicks "Enroll"
Concurrency control: locks, unique constraints, exclusion constraints	Calls to external services: payment providers, email, storage
Set-based work: joins, aggregates, filtering, sorting, pagination	Presentation: formatting prices, localizing dates, rendering pages
Tenant isolation as a last line of defense (RLS)	Authentication and the decision of who the current user is
Derived values that must always match (generated columns)	Business rules that change weekly or vary per customer

The left column holds things that must be true for *every* writer and that are cheap to check where the data lives. The right column holds things that change often, talk to the outside world, or depend on who is asking. Some rules end up in both places on purpose: the application validates a course price so it can show a friendly message on the form, and the CHECK constraint guarantees no code path ever stores a negative one. Validation in the app is for user experience; constraints in the database are for correctness. Chapter 15 returns to this boundary with views, functions, and triggers.

> **TIP**
>
> When you are unsure where a rule belongs, ask: "What happens if a second code path, written next year by someone else, forgets this rule?" If the answer is corrupted data, put the rule in the database as a constraint. If the answer is a slightly worse user experience, the application is fine.

Larkhaven in Practice: Lin's First Week

On Lin's first day, Maya asks for something that sounds simple: "When a learner enrolls, send a welcome email." Lin's first draft calls the email service directly from the request handler, right after inserting the enrollment and before the transaction commits.

Omar's review raises two problems. If the email is sent and then the transaction fails (say, the learner was already enrolled and the unique constraint fires), the learner gets a welcome email for an enrollment that does not exist. If the commit succeeds but the email service is down, the request either fails after the enrollment was saved, or the email is silently lost. Neither outcome is acceptable for a paid product.

The fix uses three things from the tour. The enrollment insert relies on `UNIQUE (user_id, course_id)` with `ON CONFLICT DO NOTHING`, so a double click cannot create two enrollments. In the *same transaction*, the code inserts a job of kind `send_welcome_email` into the `jobs` table and an event into the `outbox`. After commit, a worker claims the job with `FOR UPDATE SKIP LOCKED` and sends the email; if the email service is down, the job is retried later with backoff. Either all three rows exist or none do. Lin's handler shrinks to one function call, `enroll(conn, user_id=..., course_id=...)`, which we write in Chapter 6 and connect to the queue in Chapters 13 and 14.

Lin's takeaway from the review is worth repeating: the most valuable PostgreSQL features for applications are not the exotic ones. They are constraints and transactions, used deliberately.

What's New in PostgreSQL 18 for Developers

PostgreSQL 18 was released in September 2025, and it is the version this book targets. Most of the SQL in the book also runs on PostgreSQL 15 and later; where a feature needs 16, 17, or 18, we say so. Here are the PostgreSQL 18 changes that matter most to application developers, each run on our test server.

uuidv7()

Earlier versions could generate random (version 4) UUIDs with `gen_random_uuid()`. PostgreSQL 18 adds `uuidv7()`, which produces time-ordered UUIDs: the leading bits encode a timestamp, so new values sort after old ones and insert into B-tree indexes much like an increasing number would.