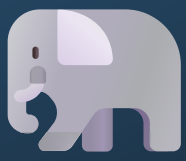

PostgreSQL

A Complete 12-Chapter Databases Course

Topics covered:

Architecture, roles, schemas & the PostgreSQL type system

JSONB, recursive CTEs, full-text search & indexing beyond B-trees

PL/pgSQL, MVCC & VACUUM, extensions (PostGIS), and replication

Exercises: 36 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Single standalone course · what's genuinely different from MySQL, not a restatement of SQL fundamentals

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 12

What Makes PostgreSQL Different — Architecture & Philosophy

PostgreSQL

Chapter 1 · What Makes PostgreSQL Different — Architecture & Philosophy

This course assumes real, working SQL knowledge — `mysql2` and `mysql3` already cover SELECT, JOINs, subqueries, CTEs, window functions, and query optimization in depth, and that knowledge transfers directly to Postgres, since both are SQL. This course is deliberately not going to re-teach any of that. Instead, every chapter asks the same question: what does Postgres actually do differently from the MySQL this site already covers, and why does it matter?

Two Relational Databases, Two Different Origins

MySQL began as MySQL AB, was acquired by Sun Microsystems in 2008, and then by Oracle in 2010 — it's now owned outright by a single corporation, dual-licensed under the GPL and a commercial license. PostgreSQL began as the POSTGRES project at UC Berkeley in 1986 under Michael Stonebraker, and is now developed by the PostgreSQL Global Development Group — a community-driven governance model with no single owning company, released under the permissive, MIT/BSD-style PostgreSQL License.

This isn't just trivia — it has real practical consequences. There's no dual-licensing tension to navigate with Postgres, no single vendor who could change licensing terms or direction unilaterally, and new features land based on community and committer consensus rather than one company's own product roadmap.

Standards-Compliance-First Design Philosophy

Postgres has historically prioritized strict SQL standard compliance and correctness over convenience shortcuts — stricter type checking, and a general unwillingness to silently coerce or truncate data rather than raising an error. MySQL's own historical trade-off leaned toward speed and ease of use, with looser default behavior in older versions (permissive date validation, silent truncation).

BE FAIR TO MODERN MYSQL

This is a common, somewhat outdated criticism of MySQL specifically. Modern MySQL versions default to strict SQL mode (`STRICT_TRANS_TABLES` and similar) and have tightened this behavior considerably since the loose-defaults era the comparison above is usually describing. The real, durable difference isn't "Postgres validates and MySQL doesn't" — both can be configured to validate strictly today. The durable difference is closer to *default philosophy*: Postgres has

> leaned toward strict correctness as its own starting point since early on, where MySQL historically optimized for ease of adoption first and tightened defaults later.

Process-Per-Connection vs. Thread-Per-Connection

Postgres gives each client connection its own operating system process — full process isolation, meaning one connection crashing can't directly take down another connection or the server as a whole. Processes are heavier to spawn than threads, though, which is exactly why connection poolers like PgBouncer become important at real scale. MySQL instead handles each client connection as a thread within a single shared server process — threads are lighter-weight to create, but a serious bug in one thread's handling puts the entire `mysqld` process genuinely at risk, since threads share memory space far more directly than isolated processes do.

This is a real architectural choice with real trade-offs, not an accident — the same category of decision `c3-3` covered when it demonstrated a genuine race condition using pthreads: shared memory between concurrent execution units buys speed at the cost of a whole class of bugs that simply can't happen when each connection is a fully separate process instead.

When to Choose Each

This isn't "Postgres is better" — both are excellent, mature, production-proven engines, and plenty of real organizations run both simultaneously for different workloads.

FAVORS MYSQL	FAVORS POSTGRESQL
Read-heavy web apps, WordPress-style CMS workloads	Complex queries, data-integrity-critical applications
Simpler replication setups	Geospatial workloads (PostGIS — previewed in this course's own Ch.10)
LAMP-stack tooling maturity and simplicity	JSON-heavy hybrid relational/document workloads (this course's own Ch.4)

This Course's Own Roadmap

DIFFERENCE NAMED IN THIS CHAPTER	RESOLVED IN
The schema concept MySQL doesn't really have	postgres1-2
A genuinely richer native type system	postgres1-3
JSONB as a document database inside a relational one	postgres1-4

DIFFERENCE NAMED IN THIS CHAPTER	RESOLVED IN
Recursive CTEs and window function extras	postgres1-5
Built-in full-text search	postgres1-6
A richer index ecosystem	postgres1-7
PL/pgSQL vs. MySQL's own stored procedures	postgres1-8
MVCC & VACUUM vs. InnoDB	postgres1-9
Extensions (PostGIS and beyond)	postgres1-10
Replication compared	postgres1-11

BASELINE ASSUMED

This course assumes the installation/administration familiarity `mysql1` already built for MySQL — `postgres1-2` covers Postgres's own installation and administration model directly, rather than starting from zero on what a database server or a client tool even is.

Hands-On Exercises

EXERCISE 1

Explain the governance/ownership difference between MySQL and PostgreSQL, and describe one concrete practical consequence of that difference.

View solution

EXERCISE 2

Explain the difference between Postgres's process-per-connection model and MySQL's thread-per-connection model, and describe one real trade-off of each.

View solution

EXERCISE 3

Given two workloads — (a) a small WordPress-based blog, and (b) an application storing complex, deeply nested product catalog data with heavy JSON usage — decide which engine this chapter's own "when to choose each" table favors for each, and justify your answer.

📄 View solution

CHAPTER 1 QUICK REFERENCE

- **MySQL** — Oracle-owned, dual-licensed · **PostgreSQL** — community-governed, permissively licensed, no single owner
- Standards-compliance-first design in Postgres vs. MySQL's historical ease-of-adoption-first defaults — though modern MySQL has tightened its own defaults considerably
- **Postgres** — process-per-connection, full isolation, heavier per-connection cost · **MySQL** — thread-per-connection, lighter-weight, shared memory risk (echoes c3-3's own pthreads race-condition material)
- Neither engine is universally "better" — real organizations run both for different workloads
- **Next chapter:** Installing & Administering PostgreSQL — psql, roles, and the schema concept MySQL doesn't really have

CHAPTER 2 OF 12

Installing & Administering PostgreSQL

PostgreSQL

Chapter 2 · Installing & Administering PostgreSQL

`postgres1-1` promised this chapter would deliver on the schema concept MySQL doesn't really have. This chapter covers that, plus the practical basics of getting a Postgres server running and administered: installation, the `psql` client, roles, and Postgres's own distinct authentication model.

Installation

On Debian/Ubuntu, Postgres installs via the standard package manager, same as `mysql1`'s own MySQL installation chapter covered. A key difference shows up immediately after install: Postgres creates a dedicated `postgres` operating-system user and a matching superuser role of the same name, and the initial database cluster is created via `initdb` — the step that actually lays down the physical storage directory Postgres will use, distinct from simply having the software installed.

psql — The Interactive Client

`psql` is Postgres's own interactive client, roughly analogous to the `mysql` command-line client — but with a genuinely different design choice: rather than special SQL-like commands for administrative tasks, `psql` uses backslash meta-commands, keeping the SQL syntax itself as close to the pure standard as possible.

TASK	PSQL (POSTGRES)	MYSQL CLIENT
List databases	`\l`	`SHOW DATABASES;`
Connect to a database	`\c dbname`	`USE dbname;`
List tables	`\dt`	`SHOW TABLES;`
Describe a table	`\d tablename`	`DESCRIBE tablename;`
List roles/users	`\du`	`SELECT user FROM mysql.user;`
Quit	`\q`	`exit`

Roles vs. Users

Postgres unifies what MySQL splits into two separate concepts. In MySQL, there are USER accounts, and a separately-evolved mechanism (roles, added only in MySQL 8.0) for grouping privileges — roles are a relatively recent addition layered on top of a user-centric model. In Postgres, there has only ever been one core concept: the **role**. A role can behave as a login-capable user, as a pure privilege-grouping "group," or both at once — the distinction comes entirely from a single attribute.

```
-- A login-capable role (behaves like a "user")
CREATE ROLE alice LOGIN PASSWORD 'secret';

-- A pure group role — cannot log in directly, only organizes privileges
CREATE ROLE reporting_team;

-- Role membership: alice inherits every privilege granted to reporting_team
GRANT reporting_team TO alice;
```

This single unifying model means privilege organization was never bolted on after the fact — it's been the same core mechanism from the start.

The Schema Concept

This is the item `postgres1-1` named as a genuine difference: a Postgres **database** contains one or more **schemas**, and each schema contains its own tables, views, and other objects — a real namespace layer sitting between "database" and "table" that MySQL simply doesn't have. In MySQL, "database" and "schema" are actually synonyms — `CREATE DATABASE` and `CREATE SCHEMA` do the literal same thing.

Every Postgres database starts with a default schema named `public` — a `CREATE TABLE` with no schema specified lands there. This becomes genuinely useful in real applications: a multi-tenant system can give each tenant its own schema within a single shared database, or a large application can separate an `app` schema from a `reporting` schema without needing entirely separate databases or connections. Postgres resolves an unqualified table name using its own `search_path` setting, checking each listed schema in order until a match is found.

pg_hba.conf — Host-Based Authentication

Postgres separates authentication into its own dedicated configuration file, `pg_hba.conf`, entirely apart from the role/privilege system itself. It controls *who* can attempt to connect, *from where*, and *using what authentication method* (`trust`, `scram-sha-256`, `peer`, `ident`, `reject`) — checked before a role's own privileges are ever evaluated at all. MySQL instead folds the authentication method directly into the user account itself (`CREATE USER 'x'@'host' IDENTIFIED BY ...`), rather than using a separate control file. Postgres's two-layer model — `pg_hba.conf` decides whether a connection attempt is even allowed, role privileges decide what it can do once connected — is a genuinely distinct architectural split.

THE CLASSIC "CORRECT PASSWORD, STILL CAN'T CONNECT" GOTCHA

A role can be created correctly, with a correct password, and still fail to connect with an authentication error — because `pg_hba.conf` has no matching rule for the connecting host or database. This is one of the most common beginner points of confusion in Postgres, precisely because the two layers (host-based rules and role privileges) are genuinely separate systems that both have to agree. Also worth knowing: a change to `pg_hba.conf` takes effect on a configuration **reload** (`pg_ctl reload` or `SELECT pg_reload_conf();`), not a full server restart — unlike some other Postgres settings.

POSTGRES1-1'S OWN ROADMAP, DELIVERED

The schema concept named as missing from MySQL back in `postgres1-1` 's own comparison table is now covered in full — every remaining chapter's own examples live inside a schema, even when that schema is just the default `public`.

Hands-On Exercises

EXERCISE 1

Explain the difference between Postgres's unified role concept and MySQL's separate user/privilege-grouping approach, with a concrete example of role membership using this chapter's own CREATE ROLE / GRANT syntax.

View solution

EXERCISE 2

Explain what a schema is in Postgres and why it doesn't have a real MySQL equivalent, and give one concrete practical use case for schemas.

View solution

EXERCISE 3

Explain the two-layer authentication model (pg_hba.conf plus role privileges), and describe this chapter's own classic gotcha — a correctly configured role that still fails to connect.

View solution

CHAPTER 2 QUICK REFERENCE

- `initdb` creates the actual storage cluster — a separate step from installing the software

- `psql` uses backslash meta-commands (`\l`, `\c`, `\dt`, `\d`, `\du`, `\q`) instead of MySQL's SHOW/DESCRIBE-style SQL commands
- **Roles** — one unified concept in Postgres (LOGIN attribute makes it "user-like"); MySQL splits users and roles, with roles added later in 8.0
- **Schemas** — a real namespace between database and table, unlike MySQL where "database" and "schema" are synonyms; default schema is `public`
- **pg_hba.conf** — a separate host-based authentication layer, checked before role privileges; changes apply on reload, not restart
- **Next chapter:** The PostgreSQL Type System — arrays, ranges, true ENUM types

CHAPTER 3 OF 12

The PostgreSQL Type System

PostgreSQL

Chapter 3 · The PostgreSQL Type System

Beyond the standard integer/varchar/date types shared with MySQL, Postgres includes several genuinely native types with no real MySQL equivalent at all. This chapter covers the four biggest ones — and sets up the distinction `postgres1-4` draws next, between structured relational richness and JSONB's own semi-structured document approach.

Arrays

Postgres allows any column to be declared as an array of any type — `integer[]`, `text[]`, and so on. A blog posts table could store its own tags directly:

```
CREATE TABLE blog_posts (
  id SERIAL PRIMARY KEY,
  title TEXT,
  tags TEXT[]
);

INSERT INTO blog_posts (title, tags)
VALUES ('Intro to Postgres', ARRAY['databases', 'postgres', 'sql']);

-- Find posts tagged 'postgres'
SELECT * FROM blog_posts WHERE tags @> ARRAY['postgres'];

-- Expand the array into individual rows
SELECT title, unnest(tags) AS tag FROM blog_posts;
```

MySQL has no native array type — the standard workaround is either a proper join table, or a comma-separated string column, a well-known anti-pattern. Postgres arrays give a genuine, indexable, queryable alternative to the string-column workaround, using containment operators (`@>, <@`) and `unnest()` — but it's still worth being honest that an array column can violate normalization the same way a comma-separated string can; it's a convenience with better tooling, not a normalization loophole.

Ranges

Range types (`int4range`, `numrange`, `tsrange`, `daterange`) represent a bounded interval as a single value. A hotel booking system can store a reservation's stay as a single `daterange`, and check for overlap directly:

```
CREATE TABLE bookings (
  id SERIAL PRIMARY KEY,
  room_id INT,
  stay DATERANGE
);

-- Does any existing booking overlap this new stay?
SELECT * FROM bookings
WHERE room_id = 12 AND stay && DATERANGE('2026-08-01', '2026-08-05');
```

That single overlap operator (`&&`) replaces hand-written boundary logic like `start1 <= end2 AND start2 <= end1`. Postgres can go further and enforce this as a genuine database-level integrity guarantee via an exclusion constraint — `EXCLUDE USING gist (room_id WITH =, stay WITH &&)` — making it structurally impossible to insert an overlapping booking for the same room at all, something with no simple MySQL equivalent.

True ENUM Types

Postgres's `CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')` creates a genuine, named, reusable type — it shows up as its own type in `\d` output, and any number of columns across any number of tables can be declared using it, sharing one single validated definition.

MySQL's ENUM is a column-level attribute, not a real named type — each column defines its own independent enum, even if two columns want identical values, with no cross-table reuse and no clean way to reference "the same enum" elsewhere. Postgres also allows adding a new value to an existing enum type cleanly (`ALTER TYPE mood ADD VALUE 'ecstatic'`), whereas altering a MySQL column-level ENUM's values typically means an actual `ALTER TABLE` against the column definition itself.

UUID & Network Types

Postgres has a genuine native `uuid` type, commonly generated with the built-in `gen_random_uuid()` (Postgres 13+) — useful for primary keys that don't leak sequential/positional information the way an auto-increment integer does. MySQL has no native UUID type; UUIDs there are typically stored as a `CHAR(36)` string or a `BINARY(16)`, both workarounds rather than a first-class type.

Postgres also has native network types — `inet` (an IP address, optionally with a subnet), `cidr` (a network specification), and `macaddr` — with real, built-in operators, like checking whether an IP falls inside a CIDR block using `<<`. MySQL has no equivalent; these would need to be stored as plain strings or integers, with all validation left entirely to application code.

RICHNESS IS A CAPABILITY, NOT AN OBLIGATION

This chapter's own type richness can tempt a design toward reaching for an array, a custom enum, or a range column when a proper join table or lookup table would actually serve the data better long-term — the same "when to choose each" judgment `postgres1-1` applied at the engine level applies again here, at the level of individual column design. A rich type system expands what's available; it doesn't change when normalization is still the right call.

STRUCTURED RICHNESS VS. SEMI-STRUCTURED DOCUMENTS

Every type in this chapter — arrays, ranges, true ENUMs — is still fully structured and relational: indexable, validated, queryable with ordinary operators. `postgres1-4` covers something categorically different: JSONB, a genuinely semi-structured, document-shaped type living inside a relational column. This chapter is "richer relational typing"; the next one is "a document database inside a relational one."

Hands-On Exercises

EXERCISE 1

Explain what a Postgres array column offers over MySQL's typical comma-separated-string workaround, and also state the real caution this chapter's own warn-box raises about overusing array/custom-type columns.

View solution

EXERCISE 2

Explain how a range type combined with an exclusion constraint solves the hotel double-booking problem this chapter describes, and why hand-written boundary-comparison logic is a weaker alternative.

View solution

EXERCISE 3

Explain the real difference between Postgres's ENUM as a genuine reusable type and MySQL's own column-level ENUM attribute, with one concrete consequence of that difference.

View solution

CHAPTER 3 QUICK REFERENCE

- **Arrays** — a native, indexable, queryable alternative to comma-separated strings, still bound by the same normalization caution

- **Ranges** — a bounded interval as one value; combined with EXCLUDE USING gist, enforces "no overlaps" at the database level
- **ENUM** — a genuine, reusable, named type in Postgres vs. MySQL's per-column attribute with no cross-table reuse
- **UUID / network types (inet/cidr/macaddr)** — first-class types with real operators, vs. MySQL's string/integer workarounds
- Richer types are a capability, not an obligation — normalization judgment still applies at the column-design level
- **Next chapter:** JSON & JSONB — a document database inside a relational one

CHAPTER 4 OF 12

JSON & JSONB — A Document Database Inside a Relational One

PostgreSQL

Chapter 4 · JSON & JSONB — A Document Database Inside a Relational One

`postgres1-3` closed by drawing a line: arrays, ranges, and true ENUMs are all still structured and relational. This chapter crosses that line — JSONB is genuinely semi-structured, document-shaped data living inside a relational column, and it's the feature that makes Postgres's own "JSON-heavy hybrid relational/document workloads" entry from `postgres1-1`'s comparison table make real sense.

JSON vs. JSONB

Postgres has two distinct JSON types. `json` stores an exact textual copy of the input — preserving whitespace, key order, and even duplicate keys — and is re-parsed on every query. `jsonb` stores a decomposed binary representation instead: no whitespace or key-order preservation, but considerably faster to query, and — critically — indexable.

Practical guidance: use `jsonb` for nearly everything. `json` is really only worth reaching for when byte-for-byte preservation of the original document matters, such as an audit log that needs to store exactly what was received.

> **BEING FAIR TO MYSQL HERE**
>
> MySQL has had its own native JSON type since 5.7.8, stored internally in an optimized binary format — genuinely comparable in spirit to `jsonb`, not a case of "MySQL has nothing." The real differentiator isn't JSON support existing at all; it's the depth of Postgres's own JSONB operator and indexing ecosystem, covered next.

JSONB Operators

OPERATOR	MEANING
`->`	Get object field or array element, returned as JSON
`->`	Get object field or array element, returned as TEXT
`#>`	Get value at a path, returned as JSON

OPERATOR	MEANING
`#>>`	Get value at a path, returned as TEXT
`@>`	Containment — does the left JSONB contain the right as a subset (the same idea as postgres1-3's own array @>)
`?`	Does this key exist at the top level?

A concrete example where this genuinely earns its keep: a product catalog where a shirt has `size` / `color` and a laptop has `ram` / `storage` — wildly different fields per category, impossible to model cleanly with fixed relational columns without either a huge, mostly-empty sparse table or an entity-attribute-value (EAV) design, both worse options in their own way.

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name TEXT,
  category TEXT,
  attributes JSONB
);

INSERT INTO products (name, category, attributes)
VALUES ('Blue T-Shirt', 'apparel', '{"size": "M", "color": "blue"}'),
       ('ThinkPad X1', 'laptop', '{"ram_gb": 32, "storage_gb": 1024}');

-- Find every laptop with at least 16GB RAM
SELECT name FROM products
WHERE category = 'laptop' AND (attributes ->> 'ram_gb')::int >= 16;
```

Indexing JSONB — GIN Indexes

A JSONB column can be indexed with a GIN (Generalized Inverted Index), making containment and key-existence queries genuinely fast at real scale:

```
CREATE INDEX idx_attrs ON products USING GIN (attributes);
```

This is the concrete payoff distinguishing Postgres's own JSONB support from a plain "store JSON as a text blob" implementation — the database can query *into* the structure with real index support, not just store and retrieve an opaque document.

Revisiting mongodb1-1's Own Document-vs-Relational Framing

`mongodb1-1` introduced the fundamental document-vs-relational distinction, and the question of when MongoDB's own model fits better than MySQL's. JSONB is the point where that clean dividing line gets genuinely blurry: a single Postgres table can have strictly relational columns — foreign keys, indexed scalars

with real integrity constraints — alongside one or more JSONB columns behaving like an embedded, schema-flexible sub-document, all in the same row, inside the same ACID-transactional engine.

This doesn't mean Postgres replaces MongoDB — a dedicated document database still has real advantages at massive horizontal scale, and a genuinely schema-less domain still fits MongoDB's own design center better. But for the extremely common real-world case of "mostly relational data, with a few genuinely variable fields," JSONB lets one Postgres database serve both needs at once, without the added complexity of keeping two separate database systems in sync.

SCHEMA FLEXIBILITY INSIDE JSONB IS A REAL TRADE-OFF, NOT A FREE LUNCH

The database can't enforce data integrity or type-checking *inside* a JSONB value the way it can for a real column — no `NOT NULL`, no `CHECK` constraint, no foreign key reference into or out of a nested JSONB field. Pushing too much of an application's actual core data model into JSONB sacrifices the very data-integrity strength `postgres1-1` credited Postgres with in the first place ("data-integrity-critical applications"). JSONB is the right tool for genuinely variable, secondary data — not a substitute for real columns on the data that matters most.

Hands-On Exercises

EXERCISE 1

Explain the difference between json and jsonb, and state this chapter's own practical guidance on which to use by default and why.

View solution

EXERCISE 2

Using the product catalog example, explain why JSONB is a better fit than either a huge sparse relational table or an EAV pattern for storing per-category product attributes.

View solution

EXERCISE 3

Explain this chapter's own honest nuance about JSONB "blurring the line" with MongoDB — what does JSONB NOT replace, and what's the real trade-off this chapter's own warn-box names about pushing core data into JSONB?

View solution

CHAPTER 4 QUICK REFERENCE

- **json** — exact textual copy, re-parsed each query · **jsonb** — decomposed binary, faster, indexable — use jsonb by default
- MySQL has had native JSON since 5.7.8 — the real gap is Postgres's operator/indexing depth, not JSON support itself
- Operators: `->` / `->>` (field access), `#>` / `#>>` (path access), `@>` (containment), `?` (key existence)
- GIN indexes make JSONB containment/key queries genuinely fast at scale — the real difference from a plain JSON text blob
- JSONB revisits mongodb1-1's document-vs-relational line — great for "mostly relational, a few variable fields," not a MongoDB replacement
- No NOT NULL/CHECK/foreign-key enforcement inside JSONB — keep core data in real columns
- **Next chapter:** Advanced Querying — Recursive CTEs & Window Function Extras

CHAPTER 5 OF 12

Advanced Querying — Recursive CTEs & Window Function Extras

PostgreSQL

Chapter 5 · Advanced Querying — Recursive CTEs & Window Function Extras

`mysql3-6` already covered window functions in real depth, and this course isn't going to re-teach ROW_NUMBER, RANK, or PARTITION BY from scratch. This chapter covers exactly two things: a feature MySQL only gained much later than Postgres (recursive CTEs), and a set of genuinely Postgres-specific window function extras that go beyond the shared ANSI SQL baseline.

Recursive CTEs

Both engines support ordinary (non-recursive) CTEs via `WITH`. A recursive CTE, using `WITH RECURSIVE`, lets a query reference itself, building up a result iteratively — the standard way to traverse a hierarchical structure (an org chart, a category tree, a bill of materials) whose depth isn't known in advance.

A recursive CTE has two parts, unioned together with `UNION ALL`: an **anchor** term (the non-recursive base case) and a **recursive** term (which references the CTE's own name and is executed repeatedly until it returns no more rows).

A Worked Example — The Employee Hierarchy

```
-- employees table has a self-referencing manager_id
WITH RECURSIVE org_chart AS (
  -- Anchor: the manager we're starting from
  SELECT id, name, manager_id, 0 AS depth
  FROM employees
  WHERE id = 7  -- the manager whose full team we want

  UNION ALL

  -- Recursive term: find direct reports of anyone already found
  SELECT e.id, e.name, e.manager_id, org_chart.depth + 1
  FROM employees e
  JOIN org_chart ON e.manager_id = org_chart.id
)
SELECT * FROM org_chart ORDER BY depth, name;
```

This returns every employee reporting to manager `7`, at any depth — direct reports, their reports, and so on — with no fixed limit on how many levels deep the hierarchy goes.

> **A REAL, MEANINGFUL GAP IN MYSQL'S OWN HISTORY**
>
> MySQL only added recursive CTE support in MySQL 8.0, released in 2018. Before that, this exact query had no equivalent in pure SQL at all — it required either application-level recursion or a stored-procedure loop. Postgres has had `WITH RECURSIVE` since version 8.4, released in 2009 — roughly a decade earlier.

Window Function Extras

Both engines support the core ANSI SQL window function specification — `mysql3-6` already covered that shared ground. This chapter covers what Postgres adds beyond it.

- **The FILTER clause** — `aggregate FILTER (WHERE condition)` is a genuinely cleaner way to do conditional aggregation than MySQL's `CASE WHEN` trick:

```
-- Postgres
SELECT COUNT(*) FILTER (WHERE status = 'completed') AS completed_count
FROM orders;

-- MySQL's equivalent
SELECT SUM(CASE WHEN status = 'completed' THEN 1 ELSE 0 END) AS completed_count
FROM orders;
```

- **Named window definitions** — a `WINDOW` clause lets a window definition be declared once and reused across several function calls in the same query, instead of repeating an identical `OVER(...)` clause each time:

```
SELECT
  name,
  RANK() OVER w AS rank,
  AVG(salary) OVER w AS dept_avg
FROM employees
WINDOW w AS (PARTITION BY department ORDER BY salary DESC);
```

- **Ordered-set aggregates** — Postgres natively supports statistical aggregates like `percentile_cont`, `percentile_disc`, and `mode()`, which MySQL doesn't provide as built-in functions at all.

Combining Recursive CTEs and Window Functions

The two features compose naturally — the employee hierarchy example above could add a window function to rank each employee's direct reports by salary within their own manager's group, applied directly on top of

the recursive result set, since a recursive CTE's output is just an ordinary result set once it's finished expanding.

A RECURSIVE CTE THAT NEVER TERMINATES IS A GENUINELY DIFFERENT FAILURE MODE

If the recursive term doesn't actually converge — a cyclic `manager_id` relationship (A reports to B, who reports back to A), or a bug in the termination logic — a recursive CTE can loop indefinitely, consuming real, growing resources rather than simply returning wrong results. This isn't an ordinary query bug: an incorrect join returns wrong-but-finite output; a non-terminating recursive CTE can hang or exhaust memory. Postgres doesn't detect cycles automatically by default (a `CYCLE` clause exists in newer Postgres versions specifically for explicit cycle detection) — worth testing recursive CTEs against known cyclic data before relying on them in production.

Hands-On Exercises

EXERCISE 1

Explain what a recursive CTE is, using this chapter's own employee hierarchy example, and explain why MySQL couldn't run an equivalent query in pure SQL before version 8.0.

View solution

EXERCISE 2

Explain the FILTER clause and, using this chapter's own example, show how it replaces MySQL's CASE WHEN conditional-aggregation trick.

View solution

EXERCISE 3

Explain this chapter's own warn-box about a recursive CTE that never terminates — what's the difference between this failure mode and an ordinary incorrect-results query bug, and what actually causes it?

View solution

CHAPTER 5 QUICK REFERENCE

- **WITH RECURSIVE** — anchor term (UNION ALL) recursive term, referencing itself until no more rows return; Postgres has had this since 8.4 (2009), MySQL only since 8.0 (2018)
- Window function basics already covered in mysql3-6 — this chapter covers Postgres-only extras
- **FILTER** — cleaner conditional aggregation than MySQL's CASE WHEN trick
- **WINDOW clause** — named, reusable window definitions across multiple function calls

- **Ordered-set aggregates** — percentile_cont/percentile_disc/mode(), no native MySQL equivalent
- A non-terminating recursive CTE hangs/consumes resources — a genuinely different failure mode than wrong-but-finite results; no automatic cycle detection by default
- **Next chapter:** Full-Text Search

CHAPTER 6 OF 12

Full-Text Search

PostgreSQL

Chapter 6 · Full-Text Search

This chapter covers a genuinely built-in Postgres capability with no dedicated MySQL comparison to lean on — full-text search — and closes with an honest look at exactly where it stops being the right tool.

Why Full-Text Search Needs Its Own Feature

Ordinary `LIKE '%word%'` matching is a poor substitute for real search: it can't be indexed efficiently for arbitrary substrings, it doesn't understand word forms (searching "running" won't match a row containing only "run"), it produces no ranking of how well a result actually matches, and it has no concept of ignoring common, low-value words like "the" or "and." Full-text search is a genuinely different capability, built specifically to solve all four problems at once.

tsvector & tsquery

A **tsvector** is a preprocessed, normalized representation of a document's searchable text — it converts raw text into *lexemes* (normalized word forms), strips out stop words, and can optionally weight different parts of a document differently (a title mattering more than body text, for instance). A **tsquery** is a processed search query, converted into the same lexeme form, supporting boolean operators: `&` (AND), `|` (OR), `!` (NOT), and `<->` (phrase/proximity).

```
SELECT to_tsvector('english', 'The runners were running quickly');
-- 'quickli':5 'run':3 'runner':2

SELECT to_tsquery('english', 'run & quick');
-- 'run' & 'quick'
```

The `@@` match operator tests whether a tsvector satisfies a tsquery, returning a simple true/false — but the real value shows up once these are combined into an actual search query.

A Worked Example

```
CREATE TABLE articles (
  id SERIAL PRIMARY KEY,
  title TEXT,
  body TEXT,
  search_vector TSVECTOR GENERATED ALWAYS AS (
    setweight(to_tsvector('english', title), 'A') ||
    setweight(to_tsvector('english', body), 'B')
  ) STORED
);

CREATE INDEX idx_articles_search ON articles USING GIN (search_vector);

-- A user-friendly search using natural query syntax
SELECT title, ts_rank(search_vector, query) AS rank
FROM articles, websearch_to_tsquery('english', 'postgres indexing') query
WHERE search_vector @@ query
ORDER BY rank DESC;
```

`setweight` makes title matches rank higher than body matches; `websearch_to_tsquery` parses ordinary user search input (rather than requiring the user to type boolean operators directly); and `ts_rank` orders results by actual relevance, not just by whether they matched at all.

THE SAME INDEX TYPE AS POSTGRES1-4'S OWN JSONB INDEXING

A tsvector column is indexed with the exact same GIN (Generalized Inverted Index) structure `postgres1-4` used for JSONB containment queries — a nice concrete demonstration of GIN's own versatility across two genuinely different feature areas.

Honest Contrast — Postgres Full-Text Search vs. a Dedicated Search Engine

Postgres's own full-text search is a real, built-in capability with no extra infrastructure to deploy — a genuine advantage for small-to-medium applications that need "good enough" search without operating a whole separate system. But dedicated search engines like Elasticsearch/OpenSearch offer real things Postgres's own full-text search doesn't attempt to match: distributed horizontal scaling across a cluster, more sophisticated relevance-tuning and language analyzers, faceted search/aggregations as a first-class feature, and typo-tolerant fuzzy matching out of the box — because search is their entire purpose, not a feature layered onto a general-purpose relational engine.

The honest guidance: Postgres's own full-text search is the right choice when search is a secondary feature of a primarily-relational application at moderate scale. A dedicated search engine is the right choice when search itself *is* the primary product, or when scale and sophistication genuinely demand it. This is the same "when to choose each" judgment from `postgres1-1`, applied once more — this time between a built-in feature and an entirely separate system.

A SINGLE FIXED LANGUAGE CONFIGURATION IS A REAL GOTCHA FOR MULTILINGUAL CONTENT

`to_tsvector('english', ...)` applies English-specific stemming and stop-word rules. If an application's content genuinely spans multiple languages, a single fixed configuration will produce poor search results for content written in any other language — this isn't handled automatically. Real multilingual full-text search needs a per-row or per-column language-aware configuration, not something Postgres provides for free out of the box.

Hands-On Exercises

EXERCISE 1

Explain why ordinary LIKE '%word%' matching is a poor substitute for real full-text search, naming at least two concrete limitations from this chapter.

View solution

EXERCISE 2

Explain the roles of tsvector and tsquery and the @@ operator, using this chapter's own worked example elements (setweight, websearch_to_tsquery, ts_rank).

View solution

EXERCISE 3

Using this chapter's own honest contrast section, explain when Postgres's built-in full-text search is the right choice, and when a dedicated search engine like Elasticsearch/OpenSearch is the right choice instead.

View solution

CHAPTER 6 QUICK REFERENCE

- LIKE '%word%' can't be indexed for arbitrary substrings, ignores word forms, produces no relevance ranking, and has no stop-word handling
- **tsvector** — normalized, lexeme-based document representation · **tsquery** — normalized, lexeme-based search query · **@@** — match operator
- `setweight` (title vs. body ranking), `websearch_to_tsquery` (user-friendly parsing), `ts_rank` (relevance ordering)
- tsvector columns are indexed with GIN — the same index type postgres1-4 used for JSONB
- Built-in search fits secondary-feature, moderate-scale needs; a dedicated engine (Elasticsearch/OpenSearch) fits when search itself is the product or scale/sophistication demands it
- A fixed language configuration (e.g. 'english') is a real gotcha for genuinely multilingual content

- **Next chapter:** Indexing Beyond B-Trees — GIN/GiST/BRIN/Hash

CHAPTER 7 OF 12

Indexing Beyond B-Trees

PostgreSQL

Chapter 7 · Indexing Beyond B-Trees

`postgres1-4` and `postgres1-6` both reached for a GIN index without fully explaining what one actually is. This chapter closes that gap, and covers the rest of Postgres's own richer index ecosystem — genuinely broader than MySQL's mostly B-tree-plus-hash approach.

B-Tree — The Default, Shared With MySQL

Both engines default to a B-tree index for ordinary equality and range queries — this is shared ground, not new material. Everything below is what Postgres adds beyond it.

GIN — Generalized Inverted Index

GIN stores a mapping from each individual *component* of a value — a JSONB key, an array element, a tsvector lexeme — to the list of rows containing it. This is a structurally different approach from a B-tree, which is built around ordering a single value per row; GIN is built for "does this composite value contain X" queries instead, which is exactly why it was used, unexplained until now, for `postgres1-4`'s own JSONB containment queries and `postgres1-6`'s own full-text search matching.

The trade-off: GIN indexes are genuinely more expensive to update than B-tree indexes — writes cost more, since a single row update can touch many entries in the index (one per component). GIN is a real cost, not a free upgrade.

GiST — Generalized Search Tree

GiST supports a broader, extensible class of queries than GIN — nearest-neighbor search, geometric and spatial queries (previewing `postgres1-10`'s own PostGIS coverage), and, notably, it's the exact index structure powering `postgres1-3`'s own `EXCLUDE USING gist` range-overlap constraint.

In general, GIN tends to be faster for lookups once built but slower to build and update; GiST is more flexible across a wider variety of query types but can be slower for simple lookups than a specialized GIN index — a genuine, real trade-off in both directions, not a case of one being strictly worse.

BRIN — Block Range Index

BRIN (Block Range Index) is designed specifically for very large tables where a column's values have a natural physical correlation with insertion order — a timestamp column on an append-only log or events table is the classic case, since rows are naturally inserted in roughly chronological order. Rather than indexing every individual row the way B-tree, GIN, and GiST all do, BRIN stores only summary information — typically the min/max values — for each physical block range of the table. The result is a dramatically smaller index, much cheaper to maintain, that still enables efficient range queries when the correlation assumption genuinely holds.

Hash Indexes

Hash indexes exist in both engines, but Postgres's own history here is worth being honest about: Postgres hash indexes weren't crash-safe or WAL-logged before Postgres 10, a real, documented limitation. They're safe to use now, but remain a genuinely narrow tool — useful only for pure equality lookups (`=`), never for range queries — and in practice, B-tree is usually still preferred, since it handles equality well *and* supports range queries too. Hash indexes remain a narrow, rarely-the-best-choice option even today.

Choosing the Right Index

INDEX TYPE	BEST FOR	NOT GOOD FOR
B-Tree	Equality and range queries on ordinary columns	Containment queries, huge tables with cheap alternatives available
GIN	JSONB containment, full-text search, array containment	Write-heavy tables (expensive to update)
GiST	Range exclusion constraints, geometric/spatial data, nearest-neighbor	Simple equality lookups better served by B-tree
BRIN	Huge, naturally-ordered tables (time-series/logs)	Data with no physical correlation to insertion order
Hash	Pure equality lookups only	Almost everything else — B-tree usually wins anyway

BRIN'S OWN CORE ASSUMPTION CAN SILENTLY BREAK

If the physical correlation a BRIN index depends on doesn't actually hold — rows bulk-loaded out of chronological order, or heavily updated/moved after insertion — a BRIN index becomes far less effective, and can even make the query planner's own choices worse than having no index at all, since the planner may still choose to use a summary index that no longer reflects reality. BRIN is only as good as the physical ordering it assumes.

NOW YOU KNOW WHAT THOSE ACTUALLY WERE

`postgres1-3`'s `EXCLUDE USING gist`, `postgres1-4`'s GIN index on JSONB, and `postgres1-6`'s GIN index on a tsvector column were all used before being formally explained — this chapter is where all three finally get their real mechanism spelled out.

Hands-On Exercises

EXERCISE 1

Explain what makes GIN indexes structurally different from B-tree indexes, and why that difference makes GIN well suited to JSONB containment and full-text search specifically.

View solution

EXERCISE 2

Explain what a BRIN index is and why it's dramatically smaller than a B-tree index on the same large table — then explain this chapter's own warn-box gotcha about when BRIN stops being effective.

View solution

EXERCISE 3

Explain the GiST index's connection to postgres1-3's own EXCLUDE USING gist range-overlap constraint — why does that specific integrity guarantee need GiST rather than a plain B-tree?

View solution

CHAPTER 7 QUICK REFERENCE

- **B-Tree** — shared default, ordering/range queries
- **GIN** — component-to-row mapping, powers JSONB containment (postgres1-4) and full-text search (postgres1-6); expensive to update
- **GiST** — extensible, flexible query types; powers postgres1-3's own EXCLUDE USING gist and previews postgres1-10's PostGIS
- **BRIN** — tiny summary index for huge, naturally-ordered tables; breaks down if physical correlation is lost
- **Hash** — equality-only, safe since Postgres 10, but narrow — B-tree usually wins anyway
- **Next chapter:** PL/pgSQL — Postgres's Procedural Language

CHAPTER 8 OF 12

PL/pgSQL — Postgres's Procedural Language

PostgreSQL

Chapter 8 · PL/pgSQL — Postgres's Procedural Language

`mysql3-8` already covered MySQL's own stored procedures, and MySQL's procedural dialect genuinely does support variables, loops, and conditionals — this isn't a "MySQL can't do this" chapter. What's genuinely different is Postgres's own procedural architecture, ergonomics, and a real security gotcha worth tying directly back to `sqli1`.

PL/pgSQL — A Real Procedural Language, Not Just SQL Extensions

PL/pgSQL adds real programming constructs — variables, `IF` / `CASE`, loops, structured exception handling — directly around SQL statements inside a function or procedure body. The genuine architectural difference from MySQL isn't procedural capability itself; it's that Postgres supports *multiple* pluggable procedural languages for writing functions — PL/pgSQL is the default and most common, but PL/Python, PL/Perl, and PL/Tcl are also available as real, first-class options. MySQL has only its own single built-in procedural SQL dialect, with no comparable multi-language extensibility.

Functions vs. Procedures

Postgres distinguishes a **function** (returns a value, usable directly inside a query — `SELECT my_func(x)`) from a **procedure** (invoked via `CALL`, able to manage its own transactions with `COMMIT` / `ROLLBACK` inside it, with no requirement to return a value). MySQL has both `CREATE FUNCTION` and `CREATE PROCEDURE` too, so this distinction itself isn't unique to Postgres — but Postgres functions tend to integrate more fluidly into ordinary SQL, callable directly inside a `SELECT` list or a `WHERE` clause exactly like a built-in function.

A Worked Example — A Function

```
CREATE FUNCTION total_order_value(order_id_param INT)
RETURNS NUMERIC AS $$
DECLARE
  total NUMERIC;
BEGIN
  SELECT SUM(quantity * unit_price) INTO total
  FROM order_items
```

```
    WHERE order_id = order_id_param;

    RETURN COALESCE(total, 0);
  END;
  $$ LANGUAGE plpgsql;

  SELECT total_order_value(42);
```

The `$$ ... $$` around the function body is **dollar-quoting** — a genuine Postgres-specific convenience letting a multi-line body be written without escaping internal quotes. It's a small but real ergonomic win: MySQL's own `DELIMITER //` convention exists specifically because semicolons inside a stored procedure body would otherwise be misread as the end of the outer `CREATE PROCEDURE` statement, forcing a temporary delimiter change and a matching `//` at the end. Dollar-quoting sidesteps that whole dance entirely.

Triggers

Postgres triggers call a separately-defined trigger *function* — written in PL/pgSQL, using a special `TRIGGER` return type and the special `NEW` / `OLD` record variables. A common real-world example: an auto-updating `updated_at` column.

```
CREATE FUNCTION set_updated_at() RETURNS TRIGGER AS $$
BEGIN
  NEW.updated_at = now();
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_set_updated_at
BEFORE UPDATE ON articles
FOR EACH ROW EXECUTE FUNCTION set_updated_at();
```

Because the trigger logic lives in a separately-defined, reusable function, that same `set_updated_at()` function can be attached to any number of other tables' own triggers without rewriting the logic each time. MySQL instead defines the trigger body directly inline inside the `CREATE TRIGGER` statement itself, one-to-one with a single trigger — a real reusability difference.

Exception Handling

PL/pgSQL supports structured exception handling with a `BEGIN ... EXCEPTION WHEN ... END` block, catching specific error conditions (like `unique_violation`) and responding to them within the function. MySQL's own `DECLARE ... HANDLER` approach is functionally comparable — both engines really do support real exception handling — the two are simply syntactically different rather than one being categorically more capable.

DYNAMIC SQL INSIDE A FUNCTION IS JUST AS INJECTABLE AS DYNAMIC SQL IN APPLICATION CODE

Building a SQL string inside a PL/pgSQL function via string concatenation and running it with `EXECUTE` is exactly as vulnerable to SQL injection as building an unparameterized query in application code — `sqli1`'s own material applies unchanged, whether the vulnerable code lives inside the application or inside the database itself. The fix is the same principle `sqli1` taught: `EXECUTE ... USING` with real parameters, rather than concatenating untrusted input directly into the SQL string.

POSTGRES1-1'S OWN ROADMAP, DELIVERED

This closes the "PL/pgSQL vs. MySQL's own stored procedures" item from `postgres1-1`'s roadmap table. Next up is this course's own central architectural chapter — MVCC and VACUUM.

Hands-On Exercises

EXERCISE 1

Explain Postgres's own multi-language procedural extensibility (PL/pgSQL, PL/Python, etc.) and why this is a genuine architectural difference from MySQL's single built-in procedural dialect, rather than "MySQL can't do stored procedures at all."

View solution

EXERCISE 2

Explain what dollar-quoting is and why it's a genuine ergonomic improvement over MySQL's DELIMITER-based approach to writing a stored procedure body.

View solution

EXERCISE 3

Using this chapter's own warn-box, explain the SQL injection risk of building dynamic SQL inside a PL/pgSQL function via string concatenation, and explain the correct fix, tying your answer to sqli1's own parameterized-query material.

View solution

CHAPTER 8 QUICK REFERENCE

- PL/pgSQL is a real procedural language, comparable to MySQL's own dialect — the real difference is Postgres's multi-language extensibility (PL/pgSQL, PL/Python, PL/Perl, PL/Tcl)

- **Functions** — return a value, usable directly in a query · **Procedures** — invoked via CALL, manage their own transactions
- **Dollar-quoting** (`$$ ... $$`) — avoids MySQL's DELIMITER dance for multi-line bodies
- **Triggers** — Postgres calls a separately-defined, reusable trigger function; MySQL defines the body inline per trigger
- Exception handling — both engines support it, syntactically different (BEGIN/EXCEPTION vs. DECLARE/HANDLER)
- Dynamic SQL via EXECUTE inside a function is exactly as injectable as application code — sqli1's parameterization lesson applies unchanged
- **Next chapter:** MVCC & VACUUM — How Postgres Actually Manages Storage

CHAPTER 9 OF 12

MVCC & VACUUM — How Postgres Actually Manages Storage

PostgreSQL

Chapter 9 · MVCC & VACUUM — How Postgres Actually Manages Storage

This is this course's own central architectural chapter — not a syntax difference, but a genuinely different physical storage strategy underneath everything covered so far.

MVCC — Multi-Version Concurrency Control

MVCC exists so that readers never block writers and writers never block readers — each transaction sees a consistent snapshot of the database as of when it started, even while other transactions concurrently modify data. To be fair from the outset: this isn't unique to Postgres. MySQL's InnoDB engine has real, solid MVCC too — a common misconception worth correcting directly rather than implying otherwise.

What genuinely differs is *how* each engine physically implements it. Every Postgres row (tuple) carries hidden system columns `xmin` and `xmax` — `xmin` records the transaction ID that created this row version, `xmax` records the transaction ID that deleted or superseded it. An `UPDATE` in Postgres never modifies a row in place — it creates an entirely new tuple with a new `xmin`, and sets `xmax` on the old tuple to mark it superseded. That old version isn't removed immediately; it becomes a **dead tuple**.

InnoDB takes a different physical approach entirely: it keeps one current row in the main table data, plus a separate **undo log** storing the information needed to reconstruct older versions for transactions that still need to see them. Postgres duplicates the row itself on every update; InnoDB keeps one row plus reconstructable history.

Dead Tuples & Why VACUUM Exists

This is the direct, structural consequence of Postgres's own "new tuple per update" strategy: every `UPDATE` and `DELETE` leaves behind dead tuples — no longer visible to any current or future transaction, but still physically occupying disk space until cleaned up.

`VACUUM`'s job is to scan a table, identify dead tuples no longer needed by any currently-running transaction, and mark that space as reusable for future inserts and updates. An important nuance: ordinary `VACUUM` does *not* shrink the file on disk — that's `VACUUM FULL`, a much heavier, table-locking operation. Ordinary `VACUUM`

just marks space as internally reusable, so the table can absorb new data without allocating new disk space, without ever actually returning space to the operating system.

If `VACUUM` falls behind — running less often than the table's own update/delete rate demands — dead tuples accumulate faster than they're reclaimed, and the table's real on-disk size can grow well beyond what its live data actually requires. This is **table bloat**, a genuine, real operational concern with no real equivalent in InnoDB's own architecture, precisely because InnoDB never creates new physical tuples on update the same way.

autovacuum

Postgres runs an `autovacuum` background process by default, automatically triggering `VACUUM` (and `ANALYZE`, which refreshes query-planner statistics) once a table crosses a configurable dead-tuple threshold — the practical, day-to-day answer to dead tuples, running automatically rather than needing manual scheduling. Tuning knobs like `autovacuum_vacuum_scale_factor` and `autovacuum_vacuum_threshold` control exactly when it triggers per table; a high-churn table often needs more aggressive tuning than the defaults provide.

Contrasted With MySQL's InnoDB

	POSTGRESQL	MYSQL (INNODB)
MVCC?	Yes, in both — real, working implementations in each	
Physical strategy	New tuple per UPDATE, old tuple marked dead	One current row + a separate undo log for older versions
Cleanup mechanism	VACUUM / autovacuum	A purge thread reclaiming undo log entries
Distinctive operational risk	Table bloat if VACUUM falls behind	Undo/history-list growth under long-running transactions

Both strategies are real, working trade-offs, not a case of one engine "having MVCC" and the other not. Postgres's strategy makes table bloat a genuine, distinctive operational concern; InnoDB avoids that specific problem, but has its own version of it — a long-running transaction can prevent old undo-log entries from being purged, causing InnoDB's own undo tablespace to grow instead. Different physical strategies, each with its own honest cost.

THE SINGLE MOST COMMON REAL TRIGGER OF SEVERE TABLE BLOAT

A single old, still-open transaction — even an idle one left open by an application bug, or a long-running analytical query — prevents `VACUUM` from cleaning up any dead tuple newer than that transaction's own snapshot, since that transaction might still legitimately need to see those older versions. Autovacuum can be running perfectly normally and this can still happen — the bottleneck isn't a lack of vacuuming, it's a transaction that's been left open far longer

than intended, silently blocking cleanup the entire time. This is the single most common real-world cause of severe table bloat in production Postgres systems.

CLOSING THE LOOP BACK TO POSTGRES1-1

`postgres1-1` 's own process-per-connection material and `c3-3` 's pthreads material both set up this chapter's real payoff: MVCC is the actual mechanism that lets many concurrent processes read a consistent view of the data without blocking each other — the storage-level answer to the concurrency model question this course opened with.

Hands-On Exercises

EXERCISE 1

Explain how Postgres physically implements MVCC using xmin/xmax and new tuple versions on UPDATE, and explain specifically why this creates dead tuples as a direct, structural consequence.

View solution

EXERCISE 2

Explain what VACUUM does — and importantly, what ordinary VACUUM does NOT do, versus VACUUM FULL — and explain "table bloat" as a consequence of VACUUM falling behind.

View solution

EXERCISE 3

Using this chapter's own warn-box, explain how a single long-running or idle-open transaction can cause severe table bloat even when autovacuum is running normally, and why VACUUM can't simply ignore that old transaction's own requirements.

View solution

CHAPTER 9 QUICK REFERENCE

- Both Postgres and InnoDB have real MVCC — the difference is physical strategy, not whether MVCC exists at all
- **Postgres** — new tuple per UPDATE (xmin/xmax), old tuple becomes a dead tuple · **InnoDB** — one current row + a separate undo log
- **VACUUM** marks dead tuple space reusable — it does NOT shrink the file (that's VACUUM FULL, table-locking)
- **Table bloat** — dead tuples accumulating faster than VACUUM reclaims them; no real InnoDB equivalent

- **autovacuum** — the automatic, default answer, tunable per table via scale_factor/threshold settings
- A single long-running/idle-open transaction is the most common real cause of severe bloat, even with autovacuum running normally
- MVCC is the actual mechanism enabling postgres1-1's own process-per-connection concurrency model to work without heavy locking
- **Next chapter:** Extensions & the Postgres Ecosystem — CREATE EXTENSION, PostGIS

CHAPTER 10 OF 12

Extensions & the Postgres Ecosystem

PostgreSQL

Chapter 10 · Extensions & the Postgres Ecosystem

This chapter delivers directly on `postgres1-1`'s own "when to choose each" table — the "Geospatial workloads (PostGIS)" entry — and closes the loop on the GiST material from `postgres1-7`.

What an Extension Actually Is

A Postgres extension is a packaged, installable addition to a running instance — new types, functions, operators, and even index access methods, bundled and installed with a single `CREATE EXTENSION extension_name;`, rather than requiring a Postgres recompile or manually loading a scattering of separate SQL and library files. Extensions register cleanly, can be removed with `DROP EXTENSION`, and are versioned (`ALTER EXTENSION ... UPDATE`).

MySQL has its own plugin mechanism, a comparable low-level C-plugin system — but nothing with quite the same reach or the "run one command, immediately gain new SQL-level types, operators, and functions" workflow. Postgres's own extension architecture is a genuinely major reason for its rich feature ecosystem — several features already used earlier in this course, like the `pgcrypto` extension for cryptographic functions, are themselves extensions rather than built-in core features.

PostGIS — The Headline Example

PostGIS is a massive, mature, industry-standard geospatial extension — genuinely one of the most-used, most mature open-source GIS systems in the world, not a toy add-on. It adds real geometry and geography types (`POINT`, `POLYGON`, `LINESTRING`), hundreds of spatial functions (`ST_Distance`, `ST_Contains`, `ST_Intersects`), and spatial indexing built directly on GiST — a direct, concrete payoff of `postgres1-7`'s own material.

```
CREATE EXTENSION postgis;

CREATE TABLE stores (
  id SERIAL PRIMARY KEY,
  name TEXT,
  location GEOGRAPHY(Point)
);
```

```
CREATE INDEX idx_stores_location ON stores USING GIST (location);

-- Find every store within 5km of a customer's location
SELECT name FROM stores
WHERE ST_DWithin(location, ST_MakePoint(-122.42, 37.77)::geography, 5000);
```

The GiST index makes this kind of "find everything within a distance" query fast at real scale — exactly the extensible, non-linear query support `postgres1-7` identified as GiST's own strength, applied here to real geospatial distance instead of range overlap.

pg_stat_statements

A genuinely different kind of extension — not a new data type or feature, but an observability tool. `pg_stat_statements` tracks execution statistics (call count, total and mean execution time, rows returned) for every distinct normalized query run against the server. Querying it, ordered by total or mean execution time, is very often the single most useful starting point for diagnosing "why is this database slow" — it surfaces the actual worst offenders directly, rather than guessing.

Unlike a typical extension, enabling it is a two-step process: it must first be added to `shared_preload_libraries` in `postgresql.conf` (a configuration change requiring a server restart), and only then can `CREATE EXTENSION pg_stat_statements` actually register it.

The Broader Ecosystem

A few other well-known extensions, briefly, to give a sense of breadth: `pgcrypto` (cryptographic functions), `uuid-ossp` (older UUID generation, largely superseded by the built-in `gen_random_uuid()` since Postgres 13), `TimescaleDB` (a genuinely production-grade time-series-optimized system, built entirely as a Postgres extension rather than a separate database), and `pg_partman` (partition management). The broader point: a huge amount of genuinely serious, production-grade functionality is built *as* Postgres extensions rather than forks or entirely separate systems — real testament to how deep the extensibility actually goes.

CREATE EXTENSION ISN'T ALWAYS A ONE-STEP OPERATION, AND ISN'T FREE OF RISK

A database user with `CREATE` privilege can typically install any extension already made available at the server/OS level by an administrator — but a genuinely new extension, not yet present on the server, requires actual filesystem-level installation by someone with server access first. "Just run CREATE EXTENSION" isn't always literally one step for an application developer without server access. It's also worth a light security note: extensions can include C code that runs with the same privileges as the Postgres server process itself — installing an extension from an untrusted source is not a purely cosmetic decision.

POSTGRES1-1'S OWN ROADMAP, DELIVERED

The "Geospatial workloads (PostGIS)" entry from `postgres1-1`'s own comparison table is now fully explained — and shown to be built directly on GiST, the same index type introduced in `postgres1-7` and used there for range-overlap exclusion constraints.

Hands-On Exercises

EXERCISE 1

Explain what CREATE EXTENSION actually does, and why Postgres's own extension architecture is a genuine, meaningful difference from MySQL's own plugin system, not just a labeling difference.

View solution

EXERCISE 2

Explain PostGIS's connection to postgres1-7's own GiST material — why does efficient "find all stores within 5km" spatial querying specifically need GiST rather than a B-tree index?

View solution

EXERCISE 3

Explain what pg_stat_statements is for, why it requires a two-step setup unlike a typical extension, and describe its practical value when diagnosing a slow database.

View solution

CHAPTER 10 QUICK REFERENCE

- **CREATE EXTENSION** — packaged types/functions/operators/index methods, installed in one command; MySQL's own plugin system has nowhere near the same SQL-level reach
- **PostGIS** — mature, industry-standard geospatial extension; spatial indexing built directly on postgres1-7's own GiST
- **pg_stat_statements** — query-level execution stats, the standard first stop for diagnosing a slow database; needs shared_preload_libraries + a restart, then CREATE EXTENSION
- Real production-grade systems (PostGIS, TimescaleDB) are built AS extensions, not separate databases — real evidence of extension-system depth
- New extensions need server-level installation first; installing from untrusted sources is a genuine security consideration (C code runs with server privileges)

- **Next chapter:** Replication & High Availability Basics

CHAPTER 11 OF 12

Replication & High Availability Basics

PostgreSQL

Chapter 11 · Replication & High Availability Basics

This chapter covers Postgres's own two genuinely distinct replication mechanisms, and closes with an honest correction of a common misconception about how Postgres's own replication compares to MySQL's.

Why Replication Matters

Three real, distinct motivations: high availability (a standby ready to take over if the primary fails), read scaling (offloading read-only queries to replicas), and disaster recovery or geographic distribution.

Streaming Replication (Physical Replication)

Every change to a Postgres database is first written to the **WAL** (Write-Ahead Log) before being applied to the actual data files — a durability mechanism this course hasn't named explicitly until now, though it underlies crash recovery generally. Streaming replication works by continuously shipping WAL records from a primary server to one or more standby servers, which replay those records to stay in sync.

This is "physical" replication — a byte-for-byte, block-level copy of the entire database cluster. A replica built this way is an exact physical copy: it can't have a different schema, can't replicate only a subset of tables, and in the classic setup is read-only.

A genuine, concrete trade-off: **asynchronous** replication (the default) is faster, but leaves a small window where data could be lost if the primary fails before a replica catches up; **synchronous** replication means a commit doesn't complete until at least one replica confirms receipt — zero data loss, at the cost of real added latency on every write.

Logical Replication

Logical replication (Postgres 10+) is a genuinely different, newer mechanism: it replicates actual row-level changes via a publish/subscribe model (`CREATE PUBLICATION` / `CREATE SUBSCRIPTION`), rather than raw physical WAL bytes. This unlocks real capability physical replication can't offer: replicating just a subset of tables rather than the whole cluster, replicating between different major Postgres versions (useful for near-zero-downtime major version upgrades), and — critically — the subscriber remains a genuinely independent,

writable database that simply happens to receive a stream of changes for the tables being replicated. This is a real, meaningfully different tool, not the same mechanism with different configuration.

Contrasted With MySQL's Own Replication Model

MySQL's own replication has historically centered on binlog-based statement or row-based replication — conceptually much closer to Postgres's own logical replication in spirit, since both work at a row/statement level rather than raw physical bytes. MySQL never really had a direct equivalent to Postgres's own byte-for-byte physical streaming replication in the same way.

It's worth correcting a common misconception directly: it's not accurate to say "Postgres invented modern replication and MySQL is behind." MySQL has had working, production-grade replication for a very long time, including modern Group Replication and InnoDB Cluster for genuine multi-primary high availability. The real difference isn't "does it work" — it's architectural: Postgres offers *both* a physical (WAL-streaming) and a logical (row-level) replication mechanism as two genuinely distinct tools for different jobs, while MySQL's own replication has centered more consistently on the logical/row-level style throughout its history.

A Basic HA Pattern

A common real setup: one primary plus one or more streaming replicas, with a tool like Patroni or repmgr handling automatic failover — promoting a replica to primary if the original fails. It's worth being honest that Postgres itself doesn't include automatic failover out of the box: a raw streaming replica setup requires manual promotion (`pg_promote()`) unless a separate HA-management tool is layered on top.

REPLICATION IS NOT A BACKUP

This echoes a principle already established for two other systems on this site — `dbsec1`'s own material and `mongodb2-5`'s own replication chapter make the identical point. A mistake or an accidental deletion on the primary replicates to every standby just as faithfully as a legitimate change does — replication protects against hardware or server failure, not against data corruption or accidental deletion. Real backups remain a separate, necessary practice no replication setup replaces.

THE WAL CLOSES A SMALL IMPLICIT GAP

The WAL — the same underlying durability mechanism introduced here — is what makes Postgres's crash recovery reliable in the first place, connecting this chapter's replication material back to `postgres1-9`'s own MVCC/VACUUM chapter: durability and concurrency both rest on the same underlying write-ahead logging foundation.

Hands-On Exercises

EXERCISE 1

Explain the difference between physical (streaming) and logical replication in Postgres, and describe one concrete scenario where logical replication's own subset/cross-version capability is specifically needed and physical replication can't do it.

View solution

EXERCISE 2

Explain the honest correction this chapter makes about MySQL's own replication history — what's the real architectural difference, and why is "Postgres invented modern replication" a misconception?

View solution

EXERCISE 3

Using this chapter's own warn-box, explain why replication is not a substitute for backups, tying your answer to the specific mechanism by which a destructive change propagates to every replica.

View solution

CHAPTER 11 QUICK REFERENCE

- **Streaming (physical) replication** — byte-for-byte, whole-cluster, ships raw WAL records; async (default, faster) vs. sync (zero data loss, more latency)
- **Logical replication** — row-level, publish/subscribe, can replicate a subset of tables and across major versions; subscriber stays independently writable
- MySQL's own replication has historically been closer to "logical" in spirit — not "behind," architecturally different
- Postgres offers both physical and logical mechanisms as genuinely distinct tools; MySQL centers on the logical/row-level style
- No automatic failover out of the box — Patroni/repmgr or manual pg_promote() required
- Replication ≠ backup — a destructive change replicates just as faithfully as a legitimate one (echoes dbsec1/mongodb2-5)
- The WAL is the same durability mechanism underlying postgres1-9's own crash-recovery reliability
- **Next chapter:** Capstone — Migrating and Extending a MySQL Database in PostgreSQL

CHAPTER 12 OF 12

Capstone: Migrating and Extending a MySQL Database in PostgreSQL

PostgreSQL

Chapter 12 · Capstone: Migrating and Extending a MySQL Database in PostgreSQL

Eleven chapters covered what's genuinely different about Postgres. This capstone combines three of them — JSONB, recursive CTEs, and PL/pgSQL — into one real, working schema, deliberately reusing ordinary relational design everywhere else, exactly the way `postgres1-1` promised this course would work.

The Scenario

Start from a small, ordinary `mysql2` / `mysql3`-style e-commerce schema — customers, categories, products, orders, order items — and reimplement it in Postgres, reaching for this course's own distinguishing features only where they provide genuine value, echoing `postgres1-3`'s own warning against reaching for richness that isn't actually earning its keep.

The Schema

```
CREATE TABLE customers (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  email TEXT UNIQUE NOT NULL
);

CREATE TABLE categories (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  parent_id INT REFERENCES categories(id),
  tax_rate NUMERIC(4,3) NOT NULL DEFAULT 0.000
);

CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  category_id INT REFERENCES categories(id),
  price NUMERIC(10,2) NOT NULL,
  attributes JSONB
);
CREATE INDEX idx_products_attributes ON products USING GIN (attributes);
```

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  customer_id INT REFERENCES customers(id),
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE order_items (
  id SERIAL PRIMARY KEY,
  order_id INT REFERENCES orders(id),
  product_id INT REFERENCES products(id),
  quantity INT NOT NULL,
  unit_price NUMERIC(10,2) NOT NULL
);
```

`customers`, `orders`, and `order_items` are ordinary relational tables — nothing here needed anything Postgres-specific. `categories` and `products` are where the interesting work happens.

Recursive CTE — Category Breadcrumbs

`categories` self-references via `parent_id`, the same hierarchical pattern `postgres1-5` introduced — but this time walking *up* toward the root, rather than *down* toward the leaves the way that chapter's own employee-hierarchy example did, building a display-ready breadcrumb path:

```
WITH RECURSIVE breadcrumb AS (
  SELECT id, name, parent_id, name::TEXT AS path
  FROM categories
  WHERE id = 15  -- e.g. "Laptops"

  UNION ALL

  SELECT c.id, c.name, c.parent_id, c.name || ' > ' || breadcrumb.path
  FROM categories c
  JOIN breadcrumb ON c.id = breadcrumb.parent_id
)
SELECT path FROM breadcrumb WHERE parent_id IS NULL;
-- 'Electronics > Computers > Laptops'
```

JSONB — Product Attributes

`products.attributes` reuses `postgres1-4`'s own product catalog pattern directly — a shirt-style product might store `{"size": "M", "color": "blue"}`, a laptop-style product `{"ram_gb": 32, "storage_gb": 1024}`, all in the same column, all queryable through the GIN index already declared on it.

A Custom PL/pgSQL Function — Order Total With Tax

This function combines `postgres1-8` 's own function-writing pattern with this chapter's own recursive category walk — for each order line, it walks up the category tree to find the nearest set tax rate, exactly the kind of composition this whole course has been building toward:

```
CREATE FUNCTION order_total_with_tax(order_id_param INT)
RETURNS NUMERIC AS $$
DECLARE
  subtotal NUMERIC := 0;
  tax_total NUMERIC := 0;
  item RECORD;
  cat_tax NUMERIC;
BEGIN
  FOR item IN
    SELECT oi.quantity, oi.unit_price, p.category_id
    FROM order_items oi
    JOIN products p ON p.id = oi.product_id
    WHERE oi.order_id = order_id_param
  LOOP
    subtotal := subtotal + (item.quantity * item.unit_price);

    WITH RECURSIVE cat_walk AS (
      SELECT id, parent_id, tax_rate FROM categories WHERE id = item.category_id
      UNION ALL
      SELECT c.id, c.parent_id, c.tax_rate
      FROM categories c
      JOIN cat_walk ON c.id = cat_walk.parent_id
    )
    SELECT tax_rate INTO cat_tax FROM cat_walk WHERE tax_rate > 0 LIMIT 1;

    tax_total := tax_total + (item.quantity * item.unit_price * COALESCE(cat_tax, 0));
  END LOOP;

  RETURN subtotal + tax_total;
END;
$$ LANGUAGE plpgsql;

SELECT order_total_with_tax(42);
```

Chapter Attribution

CAPSTONE ELEMENT	CHAPTER
Ordinary relational schema (customers/orders/order_items)	mysql2 / mysql3 (assumed baseline)
Self-referencing categories, recursive breadcrumb walk	postgres1-5
JSONB product attributes + GIN index	postgres1-4, postgres1-7

CAPSTONE ELEMENT	CHAPTER
Dollar-quoted function, FOR loop, RECORD variable	postgres1-8
Nested recursive CTE inside the function	postgres1-5, postgres1-8 (composed)

HONEST SCOPE NOTE

This capstone is a schema-redesign exercise, not a live migration walkthrough — real MySQL-to-Postgres data migration tooling (like `pgloader`) is deliberately out of scope. No replication/HA setup from `postgres1-11` is applied here, and full-text search from `postgres1-6` isn't exercised, since this particular schema's own natural fit was JSONB, recursion, and PL/pgSQL instead — not every chapter's feature needs to appear in every real schema, which is itself the point. This is also a deliberately lighter, single-engine echo of the site's own `cp1` bucket-list interest in relational-to-document conversion — a real reimplementation exercise, not the harder cross-engine MySQL-to-MongoDB problem `cp1` itself describes.

POSTGRES1-1'S OWN PROMISE, KEPT

Three tables in this schema (`customers`, `orders`, `order_items`) needed nothing Postgres-specific at all — exactly as expected, since `postgres1-1` opened this course by promising it wouldn't re-teach SQL fundamentals `mysql2`/`mysql3` already cover. Postgres's own distinguishing features were reached for only where `categories` and `products` genuinely needed them — `postgres1-3`'s own "richness is a capability, not an obligation" warning, now demonstrated at the whole-schema level.

Hands-On Exercises

EXERCISE 1

Explain why the breadcrumb recursive CTE in this chapter walks in the opposite direction from postgres1-5's own employee-hierarchy example, and explain what would need to change in the anchor/recursive terms to reverse the direction.

View solution

EXERCISE 2

Explain how order_total_with_tax() composes postgres1-8's own function-writing material with postgres1-5's own recursive CTE material, rather than being purely one or the other.

View solution

EXERCISE 3

Using this chapter's own tip-box, explain why customers/orders/order_items needed nothing Postgres-specific, and explain how this demonstrates postgres1-3's own "richness is a capability, not an obligation" warning at the whole-schema level.

View solution

CHAPTER 12 QUICK REFERENCE — COURSE COMPLETE

- Ordinary relational design (customers/orders/order_items) reused unchanged, exactly per postgres1-1's own opening promise
- Recursive CTE walks UP the category tree for breadcrumbs — the reverse direction of postgres1-5's own DOWN-walking employee example
- JSONB + GIN reuses postgres1-4/postgres1-7's own product-attributes pattern directly
- The PL/pgSQL function composes postgres1-8's own function-writing pattern WITH a nested recursive CTE — a genuine synthesis, not three separate demos
- Honest scope note: no live migration tooling, no replication/HA, no full-text search — not every feature belongs in every schema
- This closes the full 12-chapter PostgreSQL course